Curso de SQLite

Bases de datos ligeras

Salvador Pozo

http://conclase.net

Prólogo

SQLite es una biblioteca que implementa en sí misma, sin servidor, y sin necesidad de configuración, un motor de base de datos SQL transicional. El código de SQLite es de dominio público y por lo tanto libre para usarlo para cualquier propósito, comercial o privado.

Introducción

El presente manual trata sobre la integración de bases de datos **SQLite** dentro de aplicaciones C y C++. Para ello nos limitaremos a explicar algunas de las funciones y estructuras del API de **SQLite** 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.

Instalación de bibliotecas para entornos de programación basados en GCC

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.

Siguiendo lo que ya es una tradición en **Con Clase** usaremos el compilador **Mingw** y el entorno de desarrollo **Dev-C++**, **Code::Blocks** o **Zinjal** para crear nuestros programas de ejemplo.

Paquetes a instalar

En la sección de descargas de **SQLite** existen varios paquetes, aunque no todos serán necesarios para poder trabajar con este motor de bases de datos, necesitaremos descargar algunos de ellos:

- sqlite-amalgamation-x_y_z.zip: donde las letras 'x', 'y' y 'z' indican la versión de la librería. (Mientras escribo esto, la versión es 3.7.3). De este zip necesitaremos los siguientes ficheros:
  - sqlite3.h: fichero de cabecera para incluir en nuestros programas.
  - sqlite3.def: fichero de definición, contiene una lista de las funciones incluidas en la biblioteca.
- sqlite-amalgamation-x.y.z.tar.gz: se trata de los mismos contenidos, pero orientados a sistemas UNIX y LINUX.
- sqlite-x_y_z.zip: programa de línea de comandos para Windows, que nos permite acceder y modificar bases de datos.
- sqlite3-x.y.z.bin.gz: el mismo programa, para entornos Linux.
- sqlitedll-x_y_z.zip: contiene una DLL con las funciones necesarias para ejecutar SQLite, además del fichero sqlite3.def.
- sqlite-x.y.z.so.gz: contiene la librería compartida precompilada para ejecutar SQLite, necesaria en Linux.

Ni que decir tiene que sólo necesitaremos los paquetes que conciernan al sistema operativo que estemos usando.

La DLL podremos mantenerla en la misma carpeta que los ficheros ejecutables que vayamos creando, y deberemos redistribuirla junto con ellos. También se puede copiar a una carpeta a la que se tenga acceso desde el sistema (que esté incluida en el PATH), por ejemplo, "system32".

El fichero *sqlite3.h* deberá estar en la carpeta *include* del compilador. En nuestros ejemplos supondremos que está en la carpeta *include/sqlite3*.

El programa en línea de comandos puede estar en cualquier sitio, y lo usaremos para acceder a las bases de datos desde fuera de los programas, de modo que podemos crear tablas, modificarlas, o trabajar con los datos que contienen. Funciona como un intérprete SQL para bases de datos SQLite.

Usar ficheros incluidos con SQLite

Volveremos a explicar el método que usaremos para obtener el fichero que nos falta para poder acceder a la DLL de *SQLite* desde nuestro programas C/C++.

SQLite no proporciona ficheros de librería estáticas para acceder al DLL, pero podemos obtenerlos a partir del propio fichero DLL y del fichero DEF. Para hacerlo crearemos una carpeta temporal en la que copiaremos los siguientes ficheros:

- El fichero de biblioteca dinámica "dll": *sqlite3.dll*.
- El fichero de definición de biblioteca "def": *sqlite3.def*.

Y estas tres utilidades:

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

Los dos primeros ficheros ya los tenemos, y los extraeremos de los paquetes correspondientes.

El siguiente paso es crear el fichero de biblioteca estática *libsqlite.a*. Para ello necesitamos las 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 el compilador. Para comodidad, copiaremos estas

utilidades al directorio temporal de trabajo, y ejecutaremos la siguiente sentencia desde la línea de comandos:

```
C:\mysqltmp>dlltool -d sqlite3.def -D sqlite3.dll -k -l
libsqlite.a -S ./as.exe
```

La utilidad *as.exe* (un ensamblador) se invoca automáticamente desde *dlltool*. Ahora ya tenemos nuestro fichero de biblioteca estática *libsqlite.a*, y lo copiaremos al directorio *lib* del compilador.

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 conseguir ficheros de librería estática en formato ".a" a partir de ficheros DLL y DEF.

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.

Dónde usar SQLite

Extraído de la documentación de SQLite.

SQLite es diferente de la mayoría de motores de base de datos SQL, ya que su primer objetivo es ser simple:

- Fácil de administrar
- Fácil de operar
- Fácil de integrar en un programa más amplio
- Fácil de mantener y personalizar

Muchas personas usan SQLite, ya que es pequeño y rápido. Pero esas cualidades son accidentes felices. Los usuarios también encuentran que SQLite es muy fiable. La fiabilidad es una consecuencia de la simplicidad. Con menos complicaciones, hay menos cosas que puedan ir mal. Así que, sí, SQLite es pequeño, rápido y confiable, pero sobre todo, SQLite se esfuerza por ser simple.

La simplicidad en un motor de base de datos puede ser una fortaleza o una debilidad, dependiendo de lo que estamos tratando de hacer. Con el fin de lograr simplicidad, SQLite ha tenido que sacrificar otras características que algunas personas encuentran útiles, tales como alta concurrencia, control de acceso de grano fino, un rico conjunto de funciones integradas, procedimientos almacenados, funciones del lenguaje esotérico de SQL, XML y / o extensiones Java, escalabilidad de tera- o peta-bytes, y así sucesivamente. Si necesitas alguna de estas características y no te importa la complejidad añadida que conllevan, SQLite probablemente no es la base de datos que necesitas. SQLite no tiene la intención de ser un motor de base de datos empresarial. No está diseñado para competir con Oracle o PostgreSQL.

La regla básica para cuándo es apropiado utilizar SQLite es la siguiente: El uso de SQLite en situaciones donde la simplicidad de administración, implementación y mantenimiento son más importantes que las características complejas innumerables que los motores de bases de datos proporcionan. De hecho, las situaciones donde la simplicidad es la mejor opción son más comunes de lo que mucha gente piensa.

Otra forma de ver SQLite es la siguiente: SQLite no está diseñado para reemplazar a Oracle. Está diseñado para reemplazar a fopen().

Situaciones en las que SQLite funciona bien

Aplicación de formato de archivo

SQLite se ha utilizado con gran éxito como el formato de archivo en disco para aplicaciones de escritorio tales como herramientas de análisis financiero, paquetes de CAD, programas de mantenimiento, etc. La tradicional operación de "Abrir Archivo" hace un sqlite3_open() y ejecuta un BEGIN TRANSACTION para obtener acceso exclusivo al contenido. "Guardar Archivo" hace un COMMIT seguido por otro BEGIN TRANSACTION. El uso de estas operaciones garantiza que las actualizaciones del archivo son atómicas, durables, aisladas, y consistente.

Se pueden añadir triggers temporales a la base de datos para registrar todos los cambios en una tabla (temporal) que registre operaciones de deshacer/rehacer. Estos cambios se pueden reproducir de nuevo cuando el usuario presiona los botones "Deshacer" y "Rehacer". Usando esta técnica, la aplicación puede tener una profundidad ilimitada de deshacer/rehacer con un código sorprendentemente pequeño.

Los dispositivos integrados y aplicaciones

Debido a una base de datos SQLite requiere una administración pequeña o nula, SQLite es una buena opción para los dispositivos o servicios que deben trabajar solos y sin apoyo humano. SQLite es una buena opción para su uso en teléfonos móviles, PDAs, etc. También funciona bien como una base de datos integrada en aplicaciones de consumo para descargar.

Sitios Web

SQLite generalmente trabaja muy bien como motor de base de datos de sitios web de tráfico bajo o medio (es decir, el 99,9% de los sitios web). La cantidad de tráfico web que SQLite puede manejar depende, por supuesto, en gran medida de cómo utilice su base de datos el sitio web. En general, cualquier sitio que recibe menos de 100K visitas al día debería funcionar bien con SQLite. La cifra de

100K/día es un cálculo conservador, no una barrera superior. SQLite ha demostrado que puede trabajar con 10 veces esa cantidad de tráfico.

Reemplazo de archivos de disco *ad hoc*

Muchos programas utilizan fopen(), fread() y fwrite() para crear y gestionar archivos de datos en formatos de diseño propia. SQLite funciona especialmente bien como un reemplazo para estos archivos de datos ad hoc.

Bases de datos internas o temporales

Para los programas que tienen una gran cantidad de datos que deben ser tamizados y clasificados de diversas maneras, a menudo es más fácil y más rápido cargar los datos en una base de datos SQLite en memoria unido a las consultas con JOIN y cláusulas ORDER BY para extraer los datos en la forma y orden necesario en lugar de crear el código para las mismas operaciones de forma manual. Usar una base de datos SQL interna de esta manera también le da al programa mayor flexibilidad, ya que se pueden agregar nuevas columnas o índices sin tener que recodificar cada consulta.

Conjunto de datos de línea de comandos herramienta de análisis

Los usuarios experimentados de SQL puede emplear el programa de sqlite de línea de comandos para analizar conjuntos de datos diversos. Los datos en bruto pueden ser importados desde archivos CSV, a continuación, que los datos se puede separar para generar un gran número de informes. Los usos posibles incluyen análisis de sitios web de registro, el análisis de las estadísticas

deportivas, elaboración de métricas de programación y análisis de resultados experimentales.

También puede hacer lo mismo con una base de datos de profesional cliente/servidor, por supuesto. Las ventajas de usar SQLite en esta situación es que SQLite es mucho más fácil de configurar y de la base de datos resultante es un archivo único que se puede almacenar en un disquete o una memoria flash o enviar por correo electrónico a un colega.

Suplente para una base de datos profesional durante demostraciones o pruebas

Si usted está escribiendo una aplicación cliente para un motor de base de datos profesional, tiene sentido utilizar un motor de base de datos genérico que le permite conectarse a muchos tipos diferentes de motores de base de datos SQL. Tiene sentido aún mayor dar un paso más e incluir SQLite en el conjunto de bases de datos y vincular estáticamente el motor SQLite con el cliente. De esta manera el programa cliente se puede utilizar independiente con un archivo de datos SQLite para pruebas o demostraciones.

Base de datos Pedagogía

Debido a que es simple de instalar y usar (la instalación es trivial: sólo tienes que copiar el sqlite o sqlite.exe ejecutable en el equipo y ejecutarlo) SQLite es un motor de base de datos bueno para su uso en la enseñanza de SQL. Los estudiantes pueden crear fácilmente bases de datos a su gusto y puede enviar las bases de datos al profesor junto con los comentarios o calificaciones. Para estudiantes más avanzados que estén interesados en estudiar cómo se implementa un RDBMS, el código modular y bien comentado y documentado SQLite pueden servir como una buena base. Esto no quiere decir que SQLite es un modelo exacto de cómo se construyen los motores de base de datos, sino más bien que un

estudiante que entienda cómo funciona SQLite puede comprender más rápidamente los principios de funcionamiento de otros sistemas.

Extensiones experimentales del lenguaje SQL

El diseño simple y modular de SQLite hace que sea una buena plataforma para la creación de prototipos de ideas, nuevas características o experimentos del lenguaje de base de datos.

1 Desde la línea de comandos

Con cierta frecuencia podemos necesitar acceder a la base de datos desde fuera de nuestra aplicación, ya sea para crear tablas, depurar la aplicación, realizar operaciones de mantenimiento sobre la base de datos, etc.

Para esas tareas podemos usar el programa desde la línea de comandos. Basta con invocarlo usando el comando "sqlite3.exe".

```
C:\programas\ejsqlite>sqlite3
SQLite version 3.7.3
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Podemos escribir ".help" para consultar los comandos del programa, o introducir sentencias SQL para su ejecución.

Los comandos empiezan con un punto, y no terminan con el punto y coma final. Se pueden ver todos los comandos en el apéndice B, pero los más útiles son los siguientes:

- .help : Muestra la lista de comandos y sus parámetros, con una pequeña indicación de lo que hacen.
- .quit o .exit : Salir del programa en línea de comandos.
- .databases : Muestra una lista de las bases de datos abiertas actualmente. Dependiendo de las circunstancias se mostrará una base de datos princpal (main), y a veces una temporal, (temp), además de las que hayamos abierto. Si hemos iniciado el programa con un parámetro que indique una base de datos, esa base de datos pasará a ser la principal (main).
- .mode : Permite cambiar entre varios formatos de listados de salida, como csv, html, insert, line o list, entre otros.

- .tables : Muestra una lista de tablas en la base de datos principal (main).
- .schema : Muestra la sentencia CREATE TABLE usada para crear las tablas de la base de datos principal. Se puede indicar un parámetro sí sólo queremos ver el esquema de ciertas tablas.
- .indices : Muestra una lista de índices en la base de datos principal.

Veamos un ejemplo. Crearemos una base de datos a la que añadiremos dos tablas, y algunos datos. Luego usaremos esta base de datos en un programa C/C++.

Empezaremos creando la base de datos, para ello arrancamos el programa indicando un nombre de fichero que contendrá la base de datos. Por convenio, la extensión del fichero será ".db", pero puede tener cualquier extensión:

```
C:\programas\ejsqlite>sqlite3 agenda.db
SQLite version 3.7.3
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------
--------------------
0    main             C:\programas\ejsqlite\agenda.db
sqlite>
```

Ahora crearemos dos tablas, una de usuarios, que contendrá los nombres y direcciones de correo de nuestra agenda, y otra de teléfonos, que contendrá los números de teléfono de los usuarios.

```
sqlite> CREATE TABLE usuario (
   ...> claveusuario INT,
   ...> nombre VARCHAR(80),
   ...> email VARCHAR(60)
   ...> );
```

```
sqlite> .tables
usuario
sqlite> .schema
CREATE TABLE usuario (
claveusuario INT,
nombre VARCHAR(80),
email VARCHAR(60)
);
sqlite> CREATE TABLE telefono (
   ...> claveusuario INT,
   ...> numero VARCHAR(15)
   ...> );
sqlite> .tables
telefono  usuario
sqlite> .schema
CREATE TABLE telefono (
claveusuario INT,
numero VARCHAR(15)
);
CREATE TABLE usuario (
claveusuario INT,
nombre VARCHAR(80),
email VARCHAR(60)
);
sqlite>
```

Por supuesto, se trata de un ejemplo simplificado, y tiene algunas carencias de diseño, pero nos servirá para este ejemplo.

Añadamos algunos valores:

```
sqlite> INSERT INTO usuario VALUES(
   ...> 1, "Fulano", "fulano@dominio.com");
sqlite> INSERT INTO usuario VALUES(
   ...> 2, "Mengano", "mengano@dominio.com");
sqlite> SELECT * FROM usuario;
1|Fulano|fulano@dominio.com
2|Mengano|mengano@dominio.com
sqlite> INSERT INTO telefono VALUES(
   ...> 1, "(854) 444 44 44");
sqlite> INSERT INTO telefono VALUES(
   ...> 1, "(854) 444 44 55");
sqlite> INSERT INTO telefono VALUES(
   ...> 2, "(857) 555 55 54");
sqlite>
```

Finalmente, una consulta multitabla sencilla:

```
sqlite> SELECT * FROM usuario,telefono USING(claveusuario);
1|Fulano|fulano@dominio.com|(854) 444 44 44
1|Fulano|fulano@dominio.com|(854) 444 44 55
2|Mengano|mengano@dominio.com|(857) 555 55 54
sqlite>
```

Por supuesto, SQLite dispone de la mayor parte de las sentencias SQL estándar, algunas de ellas con ciertas limitaciones, pero la mayoría prefectamente funcionales. Podemos crear índices, hacer consultas, insertar, modificar o eliminar filas, borrar tablas, modificarlas, etc.

2 Tablas

En este capítulo haremos incapié en las peculiaridades de SQLite con respecto a la creación, modificación y borrado de tablas. SQLite no sigue todas las especificaciones del SQL estándar, aunque las diferencias son mínimas, conviene tenerlas en cuenta a la hora de trabajar con este motor de bases de datos.

Crear tablas

A pesar de que SQLite acepta las definiciones de columna estándar en lo que respecta a los tipos, en realidad esas definiciones son sólo una declaración de intenciones, cualquier columna de una tabla SQLite puede contener cualquier tipo de datos de cualquier longitud. De hecho, sólo existen cinco tipos de datos de columnas:

- INTEGER: un valor entero con signo.
- REAL: un valor en coma flotante.
- TEXT: una cadena de texto.
- BLOB: datos abstractos.
- NULL: el valor nulo.

Cualquier otra definición de tipo de dato se convierte a uno de estos. Por ejemplo, VARCHAR(43) equivale a TEXT, INT(11) a INTEGER, etc.

Esto significa que en una columna definida como VARCHAR(10) podremos almacenar cadenas de más de 10 caracteres sin problemas.

Por otra parte, la definición hecha al crear la tabla puede ser ignorada si se insertan datos de otro tipo.

En este ejemplo vemos como no se producen errores al insertar columnas de tipos diferentes a los indicados:

```
sqlite> CREATE TABLE prueba2 (
   ...> texto TEXT,
   ...> numero INTEGER,
   ...> real REAL);
sqlite> INSERT INTO prueba2 VALUES(23, 43.2, "hola");
sqlite> SELECT * FROM prueba2;
23|43.2|hola
sqlite>
```

En cualquier caso, no conviene abusar de esta característica de SQLite, sobre todo porque es probable que los programas que trabajen con los datos almacenados en las tablas no sean tan tolerantes con estos cambios de tipo. En lo posible, intentaremos que los datos almacenados en una columna sean del tipo adecuado.

Identificadores de fila

SQLite siempre crea una columna de tipo entero autoincrementada como identificador de fila, si no se indica otra cosa.

Podemos referirnos a esa columna con los nombres "rowid", "oid" o "_rowid_". Por ejemplo:

```
sqlite> SELECT rowid,texto,numero,real FROM prueba2;
1|23|43.2|hola
sqlite>
```

En nuestras bases de datos podremos usar ese identificador como clave para relacionar tablas o para identificar filas para cualquier propósito.

Es posible obviar este comportamiento automático, indicando una columna de tipo INTEGER que sea una clave primaria. En

realidad, lo que pasará en ese caso, es que esa columna será un alias para rowid:

```
sqlite> CREATE TABLE compra (id INTEGER PRIMARY KEY, nombre
TEXT, cantidad INTEGER);
sqlite> INSERT INTO compra (nombre,cantidad) VALUES
('Tomates',2);
sqlite> INSERT INTO compra (id,nombre,cantidad) VALUES
(4,'Patatas',5);
sqlite> SELECT * FROM compra;
1|Tomates|2
4|Patatas|5
sqlite> SELECT rowid,id FROM compra;
1|1
4|4
sqlite>
```

Tablas temporales

Es posible crear tablas temporales, para ello basta con añadir la palabra *TEMP* o *TEMPORARY* entre *CREATE* y *TABLE*. Las tablas temporales se destruyen cuando se cierra la conexión con la base de datos.

Por lo demás, podemos usar esas tablas igual que las otras, cuando los datos almacenados en ellas no tengan que ser preservados para su uso posterior, por ejemplo, en tablas para hacer cálculos, tablas intermedias entre procesos, etc.

```
sqlite> CREATE TEMP TABLE compratemp (id INTEGER PRIMARY
KEY, nombre TEXT, cantidad INTEGER);
sqlite>
```

Copiar tablas

La sintaxis de CREATE TABLE nos permite crear tablas a partir de una consulta SELECT. Esta selección puede ser cualquier

selección válida: de constantes, de una tabla, de parte de una tabla, de composiciones de tablas, etc.

```
sqlite> CREATE TEMP TABLE x AS SELECT DATE('now');
sqlite> CREATE TEMP TABLE y AS SELECT * FROM compra;
sqlite> CREATE TEMP TABLE z AS SELECT id,nombre FROM compra
WHERE id=1;
sqlite> SELECT * FROM x;
2012-03-09
sqlite> SELECT * FROM y;
1|Tomates|2
4|Patatas|5
sqlite> SELECT * FROM z;
1|Tomates
sqlite>
```

Modificar tablas

La sentencia ALTER TABLE es una de las más limitadas con respecto al estándar en SQLite. Sólo es posible cambiar el nombre de la tabla y añadir columnas. No se pueden modificar columnas, ni su nombre ni el tipo de datos que contienen, ni eliminarlas, ni modificar sus restricciones.

```
sqlite> ALTER TABLE prueba2 RENAME TO prueba3;
sqlite> .tables
prueba    prueba3   telefono  usuario
sqlite> ALTER TABLE prueba3 ADD COLUMN datos BLOB;
sqlite>
```

Eliminar tablas

Eliminar tablas es sencillo, basta con usar la sentencia DROP TABLE. Esta sentencia elimina la tabla, pero no la borra del fichero de base de datos. Si pretendemos compactar ese fichero después de eliminar una tabla con muchos datos, podremos usar la sentencia VACUUM, que vaciará del fichero de base de datos todos aquellos

que no pertenezcan a ninguna tabla, es decir, datos de tablas borradas o de registros eliminadas.

```
sqlite> DROP TABLE prueba3;
sqlite> .tables
prueba    telefono  usuario
sqlite> VACUUM;
sqlite>
```

3 Insertar, modificar y borrar

Veremos a continuación algunas de las características particulares de SQLite con respecto a las sentencias para insertar, modificar y eliminar datos.

Inserción de datos

Hay tres formatos para la sentecia INSERT. El más básico permite insertar una única fila de datos:

```
sqlite> INSERT INTO usuario (nombre,email)
VALUES('Tulano','tulano@dominio.com');
sqlite>
```

En SQLite no existe el formato extendido de INSERT, de modo que será necesario insertar las filas una a una.

Si como en este caso, en la cláusula *VALUES* se indican valores para todas las columnas, se puede omitir la lista de columnas. Cuando sea así, en la lista dentro de *VALUES* deben aparecer tantos valores como columnas existan en la tabla, y el orden de asignación será de izquierda a derecha:

```
sqlite> INSERT INTO usuario
VALUES('Sillano','sillano@dominio.com');
sqlite>
```

Si en la lista de columnas se omite alguna, a las columnas que falten se les asignará el valor por defecto, si se indicó alguno al crear la tabla, o NULL, si no se indicó ninguno.

```
sqlite> INSERT INTO usuario (nombre) VALUES('Cascasano');
sqlite> .nullvalue NULL
sqlite> SELECT * FROM usuario;
Fulano|fulano@dominio.com
Mengano|mengano@dominio.com
Tulano|tulano@dominio.com
Sillano|sillano@dominio.com
Cascasano|NULL
sqlite>
```

Nota:

Hemos usado el comando .nullvalue para que se muestre la cadena "NULL" cuando el dato a mostrar para una columna sea NULL.

Insertar desde otra tabla

Es posible hacer inserciones masivas desde selecciones, ya sean de datos de otras tablas, composiciones, etc.

```
sqlite> CREATE TEMP TABLE copia AS SELECT * FROM compra
WHERE 0;
sqlite> INSERT INTO copia SELECT * FROM compra;
sqlite> SELECT * FROM copia;
1|Tomates|2
4|Patatas|5
sqlite>
```

Resolución de conflictos

En general es posible insertar filas repetidas en una tabla, aunque según la definición de bases de datos en una relación no pueden existir tuplas iguales.

Sin embargo, si definimos una clave primaria o un índice único la tabla se convierte en una relación, y es posible que se produzcan conflictos si se intenta insertar una fila con una clave repetida.

SQLite permite manipular los conflictos de inserción mediante la opción **INSERT OR acción**, donde *acción* puede ser:

- ROLLBACK: Si estamos en el interior de una transacción y se produce un conflicto de inserción, se deshace la transacción actual.
- ABORT: Esta es la acción por defecto, si se detecta un conficto la inserción se anula, pero la transacción permanece abierta, y las acciones anteriores a la inserción se mantienen.
- REPLACE: Se sustituye la fila en conflicto por los nuevos valores.
- FAIL: Parecido a ABORT, pero no se deshacen las acciones previas llevadas a cabo por la sentencia INSERT actual que ha provocado el conflicto.
- IGNORE: Se ignora el conflicto, la inserción no se lleva a cabo y se continúa ejecutando la sentencia.

Modificar filas

Mediante la sentencia UPDATE podemos actualizar o modificar una o varias columnas de una tabla o de parte de ella.

Se puede aplicar la misma resolución de conflictos que con la sentencia INSERT, y también se puede añadir una cláusula *WHERE* para limitar el número de filas afectadas:

```
sqlite> UPDATE copia SET cantidad=cantidad*2;
sqlite> SELECT * FROM copia;
1|Tomates|4
4|Patatas|10
sqlite> UPDATE copia SET cantidad=cantidad-1 WHERE
cantidad>8;
sqlite> SELECT * FROM copia;
1|Tomates|4
4|Patatas|9
sqlite>
```

Eliminar filas

No hay mucho que decir. La sentencia DELETE se usa para eliminar filas de una tabla. Se puede indicar una cláusula *WHERE* para limitar o seleccionar qué filas se borrarán:

```
sqlite> DELETE FROM copia WHERE cantidad>5;
sqlite> SELECT * FROM copia;
3|Tomates|4
sqlite>
```

También podemos borrar todas las filas de una tabla, si omitimos la cláusula *WHERE*:

```
sqlite> DELETE FROM copia;
sqlite> SELECT * FROM copia;
sqlite>
```

4 Selección de datos

La sentencia SELECT es, con diferencia, la más compleja de SQL. Permite desde mostrar valores de constantes o funciones, a hacer consultas en una tabla o composiciones en varias tablas juntas, limitando las columnas a mostrar, renombrándolas o creándolas; limitando el número de columnas, ordenandolas, etc.

Pero la sentencia SELECT no se limita a mostrar el contenido de una tabla o parte de ella, lo que hace en realidad es crear una tabla, o más precisamente, una relación (recordemos que SQL maneja bases de datos relacionales). Es decir, el resultado de una sentencia SELECT sobre una tabla o un conjunto de tablas, es una tabla. Esa tabla resultado tendrá un número determinado de filas, y un número de columnas, cada una con su nombre.

Esta es una de las sentencias que menos diferencias tiene con el SQL estándar en SQLite, de modo que en este capítulo repasaremos un poco sus diferentes posibilidades.

Mostrar constantes y funciones

Esta es la forma más sencilla. Podemos mostrar un valor:

```
sqlite> SELECT sqlite_version();
3.7.3
sqlite>
```

O varios, separados por comas:

```
sqlite> .headers on
sqlite> SELECT round(122.32), "Hola", random();
round(122.32)|"Hola"|random()
```

```
122.0|Hola|-3071908159697118318
sqlite>
```

Nota:

Hemos usado el comando de consola `.headers on` para que se muestren las cabeceras en los listados, indicando los nombres de las columnas.

Renombrar columnas

Siempre es posible cambiar el nombre de cada columna en la tabla resultado, usando la cláusula *AS*. Así, por ejemplo, la última consulta puede ser modificada para que cada columna tenga un nombre más manejable:

```
sqlite> SELECT round(122.32) AS numero, "Hola" AS saludo,
random() AS aleatorio;
numero|saludo|aleatorio
122.0|Hola|8855590400098358693
sqlite>
```

Más adelante usaremos este mecanismo para renombrar columnas calculadas por la aplicaciones de funciones agregadas (de grupos).

Contenido de tablas

Donde se complica la sintaxis de SELECT es al mostrar el contenido de tablas y composiciones.

Limitar columnas

La principal aplicación de las sentencias SELECT es mostrar todo o parte del contenido de una tabla. Es su forma más general mostraremos todas las filas y columnas, las columnas se muestran en el orden en que fueron creadas en la sentencia CREATE TABLE. Las filas en un orden arbitrario:

```
sqlite> SELECT * FROM usuario;
Fulano|fulano@dominio.com
Mengano|mengano@dominio.com
Tulano|tulano@dominio.com
Sillano|sillano@dominio.com
Cascasano|
sqlite>
```

Podemos seleccionar algunas columnas, indicando sus nombres, y separadas con comas:

```
sqlite> SELECT rowid,nombre FROM usuario;
rowid|nombre
1|Fulano
2|Mengano
3|Tulano
4|Sillano
5|Cascasano
sqlite>
```

Podemos renombrar columnas cuando se realice la consulta.

Limitar filas

Para limitar el número de filas hay dos caminos diferentes, limitar filtrando por ciertos valores de columnas o grupos de columnas o limitar el rango de filas a mostrar.

Cláusula WHERE

La cláusula *WHERE* permite filtrar los resultados, dejando sólo aquellas filas que cumplan determinadas condiciones. Después de una cláusula *WHERE* se puede colocar cualquier expresión booleana:

```
sqlite> SELECT * FROM usuario WHERE 1;
nombre|email
Fulano|fulano@dominio.com
Mengano|mengano@dominio.com
Tulano|tulano@dominio.com
Sillano|sillano@dominio.com
Cascasano|
sqlite>
```

Aunque suele ser más útil usar expresiones que involucren a alguna columna:

```
sqlite> SELECT * FROM usuario WHERE nombre LIKE "%lano";
nombre|email
Fulano|fulano@dominio.com
Tulano|tulano@dominio.com
Sillano|sillano@dominio.com
sqlite>
```

Este último ejemplo muestra una relación con las filas de usuarios cuyo nombre termina en "lano".

Cláusula HAVING

El uso de esta cláusula es más esquivo. Sirve para lo mismo que *WHERE*, pero se aplica sobre columnas creadas con funciones agregadas. Para ver su uso crearemos y poblaremos una nueva tabla:

```
sqlite> CREATE TABLE pedidos (
   ...> fecha DATE,
   ...> referencia TEXT,
   ...> cantidad INTEGER);
```

```
sqlite> INSERT INTO pedidos VALUES(date('now', '-10 days'),
123, 321);
sqlite> INSERT INTO pedidos VALUES(date('now', '-10 days'),
125, 120);
sqlite> INSERT INTO pedidos VALUES(date('now', '-10 days'),
132, 100);
sqlite> INSERT INTO pedidos VALUES(date('now', '-5 days'),
123, 95);
sqlite> INSERT INTO pedidos VALUES(date('now', '-5 days'),
132, 174);
sqlite> INSERT INTO pedidos VALUES(date('now', '-5 days'),
154, 98);
sqlite> INSERT INTO pedidos VALUES(date('now', '-5 days'),
178, 150);
sqlite> INSERT INTO pedidos VALUES(date('now', '-2 days'),
125, 50);
sqlite> INSERT INTO pedidos VALUES(date('now', '-2 days'),
126, 150);
sqlite> INSERT INTO pedidos VALUES(date('now', '-2 days'),
132, 250);
sqlite> INSERT INTO pedidos VALUES(date('now', '-2 days'),
178, 140);
sqlite> INSERT INTO pedidos VALUES(date('now', '-2 days'),
180, 100);
sqlite> SELECT * FROM pedidos;
fecha|referencia|cantidad
2012-03-13|123|321
2012-03-13|125|120
2012-03-13|132|100
2012-03-18|123|95
2012-03-18|132|174
2012-03-18|154|98
2012-03-18|178|150
2012-03-21|125|50
2012-03-21|126|150
2012-03-21|132|250
2012-03-21|178|140
2012-03-21|180|100
sqlite>
```

Ahora haremos una cosulta en la que sumaremos las columnas 'cantidad' para cada 'referencia' distinta:

```
sqlite> SELECT referencia,sum(cantidad) FROM pedidos GROUP
```

```
BY referencia;
referencia|sum(cantidad)
123|416
125|170
126|150
132|524
154|98
178|290
180|100
sqlite>
```

Si queremos limitar el número de filas a las que la suma de cantidades sea menor de 160 no podremos usar la cláusula *WHERE*:

```
sqlite> SELECT referencia,sum(cantidad) FROM pedidos WHERE
sum(cantidad)<160 GROUP BY referencia;
Error: misuse of aggregate: sum()
sqlite>
```

En su lugar debemos usar *HAVING*:

```
sqlite> SELECT referencia,sum(cantidad) FROM pedidos GROUP
BY referencia HAVING sum(cantidad)<160;
referencia|sum(cantidad)
126|150
154|98
180|100
sqlite>
```

O, renombrando la columna de la suma:

```
sqlite> SELECT referencia,sum(cantidad) AS suma FROM pedidos
GROUP BY referencia HAVING suma<160;
referencia|suma
126|150
154|98
180|100
sqlite>
```

Cláusula LIMIT

Por último, la cláusula *LIMIT* nos permite limitar el número de filas de la rabla de salida:

```
sqlite> SELECT * FROM pedidos LIMIT 4;
fecha|referencia|cantidad
2012-03-13|123|321
2012-03-13|125|120
2012-03-13|132|100
2012-03-18|123|95
sqlite>
```

La cláusula opcional *OFFSET* permite indicar un desplazamiento, por ejemplo, para obtener las cuatro filas a partir de la tercera, indicaremos un límite de 4 con un desplazamiento de 2:

```
SELECT * FROM pedidos LIMIT 4 OFFSET 2;
fecha|referencia|cantidad
2012-03-13|132|100
2012-03-18|123|95
2012-03-18|132|174
2012-03-18|154|98
```

Si no se especifica una cláusula *ORDER BY* el orden es el mismo en que se almacenan en la tabla de origen.

Ordenar las filas

Mediante la cláusula *ORDER BY* podemos establecer distintos criterios para ordenar las filas. A continuación de la cláusula se especifica un nombre de columna o varios, separados por comas. Para cada columna podemos elegir un orden ascendente *ASC*, que es el orden por defecto; o descendente *DESC*:

```
sqlite> SELECT * FROM pedidos ORDER BY fecha DESC;
fecha|referencia|cantidad
```

```
2012-03-21|125|50
2012-03-21|126|150
2012-03-21|132|250
2012-03-21|178|140
2012-03-21|180|100
2012-03-18|123|95
2012-03-18|132|174
2012-03-18|154|98
2012-03-18|178|150
2012-03-13|123|321
2012-03-13|125|120
2012-03-13|132|100
sqlite> SELECT * FROM pedidos ORDER BY fecha DESC, cantidad
ASC;
fecha|referencia|cantidad
2012-03-21|125|50
2012-03-21|180|100
2012-03-21|178|140
2012-03-21|126|150
2012-03-21|132|250
2012-03-18|123|95
2012-03-18|154|98
2012-03-18|178|150
2012-03-18|132|174
2012-03-13|132|100
2012-03-13|125|120
2012-03-13|123|321
sqlite>
```

Agrupar filas

Usando la cláusula *GROUP BY* podemos agrupar filas por una o varias columnas, separadas con comas:

```
sqlite> SELECT * FROM pedidos GROUP BY referencia;
fecha|referencia|cantidad
2012-03-18|123|95
2012-03-21|125|50
2012-03-21|126|150
2012-03-21|132|250
2012-03-18|154|98
2012-03-21|178|140
2012-03-21|180|100
sqlite> SELECT fecha,count(*) AS items FROM pedidos GROUP BY
```

```
fecha;
fecha|items
2012-03-13|3
2012-03-18|4
2012-03-21|5
sqlite>
```

Por supuesto, si queremos filtrar los resultados por algún criterio, hay que tener en cuenta que esta última consulta es agregada, y por lo tanto, hay que usar la cláusula *HAVING*.

```
sqlite> SELECT fecha,count(*) AS items FROM pedidos GROUP BY
fecha HAVING items>3;
fecha|items
2012-03-18|4
2012-03-21|5
sqlite>
```

Eliminar filas duplicadas

En toda relación (o tabla) en una base de datos relacional, no debe haber filas repetidas. Esto es parte de la definición de las bases de datos relacionales. La sentencia SELECT permite eliminar filas duplicadas añadiendo la cláusula *DISTINT*:

```
sqlite> SELECT DISTINCT referencia FROM pedidos;
referencia
123
125
126
132
154
178
180
sqlite>
```

5 Operadores

Repasemos ahora los operadores SQL, y algunos específicos de SQLite.

Operadores aritméticos

Los operadores aritméticos se aplican a valores numéricos. SQLite dispone de los operadores aritméticos habituales:

*
: Multiplicación

/
: División

+
: Suma

-
: Resta

%
: Módulo o resto de división entera

Además de los operadores unitarios de signo:

+
: No hace nada

-
: Cambio de signo

Operadores booleanos

Para expresiones lógicas disponemos de los siguientes operadores, los tres de siempre:

AND
: Y lógico

OR
: O lógico

NOT
: Negación lógica

El operador NOT, por supuesto, es un operador unitario.

No existen las constantes booleanas true y false. El valor 0 se considera falso, y cualquier valor distinto de 0 se considera verdadero:

```
sqlite> SELECT 1 OR 3;
1 OR 3
1
sqlite> SELECT 0 AND 3;
0 AND 3
0
sqlite> SELECT NOT 2;
NOT 2
0
sqlite>
```

Tampoco existen, como pasa en MySQL, las alternativas &&, || y ! para los operadores AND, OR y NOT, respectivamente.

Operadores de comparación

Para hacer comparaciones disponemos del arsenal completo de operadores de comparación:

<
: Menor que

<=
: Menor o igual que

>
: Mayor que

>=

Mayor o igual que

= ó ==

Igual a

!= ó <>

Distinto de

Además de dos específicos para comparar con NULL:

IS

Si el valor *es* NULL devuelve true

NOT IS

Si el valor no es NULL devuelve true

Operadores de bits

Los operadores de bits incluyen las operaciones básicas, Y, O y No, además de los operadores de desplazamiento a izquierda y derecha.

&

Y de bits

|

O de bits

~

Complemento a 1, negación de bits

<<

Desplazamiento a la izquierda (multiplicar por 2 n veces)

>>

Desplazamiento a la derecha (división por 2 n veces)

```
sqlite> SELECT 1 | 2;
1 | 2
3
sqlite> SELECT 1 & 3;
1 & 3
1
sqlite> SELECT 1 << 3;
1 << 3
```

```
8
sqlite> SELECT 8 >> 2;
8 >> 2
2
sqlite> SELECT ~5;
~5
-6
sqlite>
```

Concatenación

SQLite dispone de un operador para concatenar cadenas:

||
Concatenación

```
sqlite> SELECT "Hola" || " y " || "adios";
"Hola" || " y " || "adios"
Hola y adios
sqlite>
```

Pertenencia a conjunto

El operadro *IN* permite verificar si un valor está dentro de un conjunto dado:

```
sqlite> SELECT 'a' IN ('a','e','i','o','u');
'a' IN ('a','e','i','o','u')
1
sqlite>
```

Comparación con patrones

Disponemos de dos operadores para camparar cadenas con un patrón:

LIKE
: Operador de patrones SQL, los comodines son % para cadenas y _ para caracteres, no distingue mayúsculas de minúsculas

GLOB
: Operador de patrones SQLite, los comodines son * para cadenas y ? para carácteres, se distingue entre mayúsculas y minúsculas

```
sqlite> SELECT 'hola' LIKE '_o%', 'hola' GLOB 'H*';
'hola' LIKE '_o%'|'hola' GLOB 'H*'
1|0
sqlite>
```

Pertenencia a rango

Por último, disponemos del operador de rango *BETWEEN*, que permite verificar si un valor está incluído en un rango definido por un valor máximo y un valor mínimo.

El rango no tiene por qué ser de números:

```
sqlite> SELECT 2 BETWEEN 0 AND 4;
2 BETWEEN 0 AND 4
1
sqlite> SELECT 'fa' BETWEEN 'aa' AND 'zz';
'fa' BETWEEN 'aa' AND 'zz'
1
sqlite>
```

6 Restricciones de columna

Ya hemos usado algunas, y ahora las repasaremos y veremos el resto.

Generalmente, las restricciones limitan los valores de las columnas, de modo que se asjuste a ciertas normas.

DEFAULT

La restricción *DEFAULT* establece un valor por defecto para una columna. Ese valor se usará cuando se inserten filas sin especificar un valor concreto para la columna:

```
sqlite> CREATE TABLE t(
   ...> nombre TEXT DEFAULT "desconocido",
   ...> cantidad INTEGER DEFAULT 0
   ...> );
sqlite> INSERT INTO t(nombre) VALUES("Fernando");
sqlite> INSERT INTO t(cantidad) VALUES(10);
sqlite> SELECT * FROM t;
nombre|cantidad
Fernando|0
desconocido|10
sqlite>
```

NOT NULL

La restricción *NOT NULL* impide que una columna pueda tomar el valor NULL. Cualquier intento de asignar un valor nulo a la columna producirá un error:

```
sqlite> CREATE TABLE t1 (c1 TEXT NOT NULL);
sqlite> INSERT INTO t1 (c1) VALUES(NULL);
```

```
Error: t1.c1 may not be NULL
sqlite>
```

UNIQUE

La restricción *UNIQUE* impide que existan dos filas con el mismo valor para la columna indicada.

```
sqlite> CREATE TABLE t2 (c1 TEXT UNIQUE, c2 INTEGER);
sqlite> INSERT INTO t2(c1,c2) VALUES("Patatas", 2);
sqlite> INSERT INTO t2(c1,c2) VALUES("Tomates", 6);
sqlite> INSERT INTO t2(c1,c2) VALUES("Pimientos", 4);
sqlite> INSERT INTO t2(c1,c2) VALUES("Tomates", 2);
Error: column c1 is not unique
sqlite>
```

Una restricción *UNIQUE* implica la creación de un índice sobre la columna a la que se aplica.

Una columna con esta restricción puede contener el valor *NULL*.

```
sqlite> INSERT INTO t2(c1,c2) VALUES(NULL, 9);
sqlite> INSERT INTO t2(c1,c2) VALUES(NULL, 15);
sqlite> .nullvalue "NULL"
sqlite> SELECT * FROM t2;
c1|c2
Patatas|2
Tomates|6
Pimientos|4
TOmates|2
NULL|9
NULL|15
sqlite>
```

Nota:

El comando de consola *.nullvalue* permite definir una cadena para mostrar valores *NULL*.

PRIMARY KEY

La restricción *PRIMARY KEY* limitan los valores de la columna de varias formas:

- Se crea un índice sobre la columna a la que se aplica la restricción.
- Se aplica la misma restricción que con *UNIQUE*, es decir, no pueden existir dos filas con el mismo valor para una columna a la que se aplica la restricción.
- Se aplica la restricción *NOT NULL*.

Además, en el caso de SQLite, si la columna es entera, se asume la cláusula *AUTOINCREMENT*.

```
sqlite> CREATE TABLE t3 (id INTEGER PRIMARY KEY, nombre
TEXT);
sqlite> INSERT INTO t3 (nombre) VALUES("Pedro");
sqlite> INSERT INTO t3 (nombre) VALUES("Antonio");
sqlite> INSERT INTO t3 (nombre) VALUES("Carlos");
sqlite> SELECT * FROM t3;
id|nombre
1|Pedro
2|Antonio
3|Carlos
sqlite>
```

FOREIGN KEY

Mediante la restricción *FOREIGN KEY* se vincula el valor de una columna con el de otra en otra tabla diferente, llamada tabla *padre*. Es decir, una columna con esta restricción sólo puede tomar valores que existan en la columna vinculada de la tabla padre. A esa columna se le denomina clave foránea.

Para que SQLite soporte claves foráneas debe estár compilado con las opciones adecuadas, y además se debe activar el *PRAGMA* foreign_keys:

```
sqlite> PRAGMA foreign_keys = ON;
sqlite>
```

Crearemos ahora dos tablas vinculadas con una clave foránea:

```
sqlite> CREATE TABLE artista(
   ...> idartista INTEGER PRIMARY KEY,
   ...> nombre TEXT
   ...> );
sqlite> CREATE TABLE cancion(
   ...> idcancion INTEGER PRIMARY KEY,
   ...> titulo TEXT,
   ...> idartista INTEGER REFERENCES artista(idartista)
   ...> );
sqlite> INSERT INTO artista (nombre) VALUES("Mike
Oldfield");
sqlite> INSERT INTO artista (nombre) VALUES("The Beatles");
sqlite> SELECT * FROM artista;
idartista|nombre
1|Mike Oldfield
2|The Beatles
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Tubular Bells", 1);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Yellow Submarine", 2);
sqlite> INSERT INTO cancion (titulo,idtista) VALUES("Get
Back", 2);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Yesterday", 2);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Sympathy For The Devil", 3);
Error: foreign key constraint failed
sqlite> SELECT * FROM cancion;
idcancion|titulo|idartista
1|Tubular Bells|1
2|Yellow Submarine|2
3|Get Back|2
4|Yesterday|2
sqlite>
```

En este ejemplo, cada canción debe tener un autor, que se indica añadiendo una clave de la tabla *artista*. La restricción de clave foránea impide asignar un valor de identificador de artista que no exista.

Acciones ON UPDATE y ON DELETE

Pero, ¿qué pasa si borramos un artista o modificamos su identificador? Esto puede provocar una violación de restricción de clave foránea. De hecho, eso es lo que ocurrirá tal como están creadas las tablas anteriores:

- No podremos eliminar filas de la tabla de artistas si existen canciones que usen su identificador de artista.
- No podremos modificar el identificador de artista si existen canciones que usen ese identificador.

SQL permite definir acciones que se tomarán automáticamente en el caso de que se eliminen o modifiquen valores que se usen como claves foráneas en otras tablas. Para ello se añaden las cláusulas opcionales *ON DELETE* y *ON UPDATE*. Para cada una se puede tomar una de las siguientes acciones:

NO ACTION
: No hacer nada, es decir, dejar las claves foráneas sin modificar. Esto, evidentemente, poduce violaciones de la restricción de clave foránea. Se supone que el usuario o la aplicación que usa la base de datos tomará las acciones necesarias para mantener las restricciones.

RESTRICT
: Indica que está prohibido hacer la modificación o borrado de la clave, si eso va a provocar una violación de restricción. Esto se parece a la acción por defecto, pero la diferencia es que no se espera a que termine la ejecución de la sentencia o de la transacción.

SET NULL

Cualquier columna en la tabla hija que haga referencia a la clave modificada o borrada se le asignará el valor NULL.

SET DEFAULT

Cualquier columna en la tabla hija que haga referencia a la clave modificada o borrada se le asignará el valor por defecto.

CASCADE

La acción se propaga a las claves de las tablas hijas. Si es una modificación, los valores de la columna en la tabla hija se modifican al mismo valor. Si es un borrado, las filas de las tablas hijas se borran.

Borremos las tablas anteriores y empecemos de nuevo:

```
sqlite> DROP TABLE cancion;
sqlite> DROP TABLE artista;
sqlite> CREATE TABLE artista(
   ...> idartista INTEGER PRIMARY KEY,
   ...> nombre TEXT
   ...> );
sqlite> CREATE TABLE cancion(
   ...> idcancion INTEGER PRIMARY KEY,
   ...> titulo TEXT,
   ...> idartista INTEGER REFERENCES artista(idartista) ON
UPDATE CASCADE ON DELETE SET NULL
   ...> );
sqlite> INSERT INTO artista (nombre) VALUES("Mike
Oldfield");
sqlite> INSERT INTO artista (nombre) VALUES("The Beatles");
sqlite> SELECT * FROM artista;
idartista|nombre
1|Mike Oldfield
2|The Beatles
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Tubular Bells", 1);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Yellow Submarine", 2);
sqlite> INSERT INTO cancion (titulo,idartista) VALUES("Get
Back", 2);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Yesterday", 2);
sqlite> SELECT * FROM cancion;
```

```
idcancion|titulo|idartista
1|Tubular Bells|1
2|Yellow Submarine|2
3|Get Back|2
4|Yesterday|2
sqlite> UPDATE artista SET idartista=3 WHERE idartista=1;
sqlite> SELECT * FROM cancion;
idcancion|titulo|idartista
1|Tubular Bells|3
2|Yellow Submarine|2
3|Get Back|2
4|Yesterday|2
sqlite> DELETE FROM artista WHERE idartista=3;
sqlite> SELECT * FROM artista;
idartista|nombre
2|The Beatles
sqlite> SELECT * FROM cancion;
idcancion|titulo|idartista
1|Tubular Bells|NULL
2|Yellow Submarine|2
3|Get Back|2
4|Yesterday|2
sqlite>
```

Vemos que las modificaciones en el identificador de artista se propagan a la tabla hija, y que cuando se borran artistas, las claves en la tabla hija toman el valor NULL.

CHECK

La restricción *CHECK* se usa para verificar si el valor de una columna está en el dominio especificado. Esto impide que se asignen valores no válidos a determinadas columnas.

```
sqlite> CREATE TABLE t4(
   ...> nombre TEXT,
   ...> cantidad INTEGER CHECK(cantidad>0)
   ...> );
sqlite> INSERT INTO t4(nombre,cantidad) VALUES("Tornillo
M3", 32);
sqlite> INSERT INTO t4(nombre,cantidad) VALUES("Tornillo
M4", 12);
```

```
sqlite> INSERT INTO t4(nombre,cantidad) VALUES("Tornillo
M5", 18);
sqlite> INSERT INTO t4(nombre,cantidad) VALUES("Tornillo
M6", 0);
Error: constraint failed
sqlite>
```

La verificación también se hace al actualizar las filas:

```
sqlite> UPDATE t4 SET cantidad=cantidad-15;
Error: constraint failed
sqlite> SELECT * FROM t4;
nombre|cantidad
Tornillo M3|32
Tornillo M4|12
Tornillo M5|18
sqlite>
```

7 Composiciones

Entendemos por composición cualquier consulta multitabla. En SQL hay varios tipos de composiciones, aunque no todos están disponibles en SQLite.

Producto cartesiano

Consiste en combinar todas las filas de todas las tablas que intervienen en la operación. Si se componen dos tablas, T1 y T2, donde T1 tiene N1 filas y M1 columnas y T2 N2 filas y M2 columnas, el resultado será una tabla de N1*N2 filas y M1+M2 columnas.

Como estas operaciones se pueden asociar, si intervienen tres tablas, la tabla resultante tendrá N1*N2*N3 filas, y M1+M2+M3 columnas.

Borremos las tablas del ejemplo anterior y reconstruyámoslas de nuevo:

```
sqlite> DROP TABLE cancion;
sqlite> DROP TABLE artista;
sqlite> CREATE TABLE artista(
   ...> idartista INTEGER PRIMARY KEY,
   ...> nombre TEXT
   ...> );
sqlite> CREATE TABLE cancion(
   ...> idcancion INTEGER PRIMARY KEY,
   ...> titulo TEXT,
   ...> idartista INTEGER REFERENCES artista(idartista) ON
UPDATE CASCADE ON DELETE SET NULL
   ...> );
sqlite> INSERT INTO artista (nombre) VALUES("Mike
Oldfield");
sqlite> INSERT INTO artista (nombre) VALUES("The Beatles");
sqlite> INSERT INTO cancion (nombre,cancionartista)
VALUES("Tubular Bells", 1);
Error: table cancion has no column named cancionartista
```

```
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Tubular Bells", 1);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Yellow Submarine", 2);
sqlite> INSERT INTO cancion (titulo,idartista) VALUES("Get
Back", 2);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Yesterday", 2);
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Moonlight Shadow",1);
sqlite> SELECT * FROM artista,cancion;
idartista|nombre|idcancion|titulo|idartista
1|Mike Oldfield|1|Tubular Bells|1
1|Mike Oldfield|2|Yellow Submarine|2
1|Mike Oldfield|3|Get Back|2
1|Mike Oldfield|4|Yesterday|2
1|Mike Oldfield|5|Moonlight Shadow|1
2|The Beatles|1|Tubular Bells|1
2|The Beatles|2|Yellow Submarine|2
2|The Beatles|3|Get Back|2
2|The Beatles|4|Yesterday|2
2|The Beatles|5|Moonlight Shadow|1
sqlite>
```

Composiciones internas

Se denominan composiciones internas aquellas en las que no aparecen filas que no estén presentes en el producto cartesiano.

Generalmente, los productos cartesianos tienen una utiliad limitada. Lo normal es que se espeficifique alguna condición para eliminar algunas de las filas de ese producto, dejando sólo las que tengan sentido.

Aunque hay muchas opciones a la hora de elegir cláusulas para hacer composiciones, en realidad casi todas son equivalentes. *JOIN*, *CROSS JOIN*, *INNER JOIN* y la coma producen los mismos resultados. La única diferencia está en *CROSS JOIN* que mantiene el orden en que se componen las tablas, mientras que el resto de las cláusulas dejan al optimizador de código que elija el orden que produzca la salida más eficientemente.

Así, las siguientes consultas tienen la misma salida:

```
sqlite> SELECT * FROM artista,cancion;
sqlite> SELECT * FROM artista INNER JOIN cancion;
sqlite> SELECT * FROM artista CROSS JOIN cancion;
sqlite> SELECT * FROM artista JOIN cancion;
```

Limitar filas de composiciones

Disponemos de varias formas de especificar condiciones que eliminen filas del producto cartesiano.

Por supuesto, está la cláusula *WHERE*:

```
sqlite> SELECT * FROM artista,cancion WHERE
artista.idartista=cancion.idartista;
idartista|nombre|idcancion|titulo|idartista
1|Mike Oldfield|1|Tubular Bells|1
2|The Beatles|2|Yellow Submarine|2
2|The Beatles|3|Get Back|2
2|The Beatles|4|Yesterday|2
1|Mike Oldfield|5|Moonlight Shadow|1
sqlite>
```

Pero esta forma tiene algunos inconvenientes.

- Se siguen generando tantas columnas como la suma de columnas de cada tabla, aunque evidentemente, en este caso las dos columnas 'idartista' son idénticas, y podría suprimirse una.
- Si necesitamos incluir más condiciones, no es evidente cuales de ellas definen la composición y cuales filtran los resultados. Esto es más bien una cuestión de estilo.
- Las cláusulas *WHERE* se evalúan para cada fila de la composición, al contrario de las que veremos ahora, que actúan antes, evitando que se generen filas que no cumplan la condición.

Las cláusulas *ON* y *USING* están orientadas específicamente a hacer composiciones.

La cláusula *ON* permite definir una condición, que puede ser cualquier expresión booleana:

```
sqlite> SELECT * FROM artista,cancion
ON(artista.idartista=cancion.idartista);
idartista|nombre|idcancion|titulo|idartista
1|Mike Oldfield|1|Tubular Bells|1
2|The Beatles|2|Yellow Submarine|2
2|The Beatles|3|Get Back|2
2|The Beatles|4|Yesterday|2
1|Mike Oldfield|5|Moonlight Shadow|1
sqlite>
```

Vemos que la salida es la misma que con *WHERE*, y que la columna 'idartista' se muestra dos veces.

Si, como en este caso, hemos usado el mismo identificador para la clave foránea en ambas tablas, podemos hacer uso de la cláusula *USING*:

```
sqlite> SELECT * FROM artista,cancion USING(idartista);
idartista|nombre|idcancion|titulo
1|Mike Oldfield|1|Tubular Bells
2|The Beatles|2|Yellow Submarine
2|The Beatles|3|Get Back
2|The Beatles|4|Yesterday
1|Mike Oldfield|5|Moonlight Shadow
sqlite>
```

Ahora sólo se muestra una vez la columna 'idartista', aunque la tabla resultante tiene la misma información.

Este tipo de composición se conoce como **composición natural**, y es tan frecuente que disponemos de una cláusula especial para crear este tipo de composiciones, *NATURAL JOIN*:

```
sqlite> SELECT * FROM artista NATURAL JOIN cancion;
idartista|nombre|idcancion|titulo
1|Mike Oldfield|1|Tubular Bells
2|The Beatles|2|Yellow Submarine
2|The Beatles|3|Get Back
```

```
2|The Beatles|4|Yesterday
1|Mike Oldfield|5|Moonlight Shadow
sqlite>
```

Nota:

Hay que tener especial cuidado con las composiciones naturales, ya que se buscarán todas las columnas en todas las tablas con el mismo nombre, y sólo se mostrarán las filas resultantes para las que los valores de todas ellas sean iguales.

Composiciones externas

En las composiciones externas sí pueden aparecer filas que no estarán presentes en un producto cartesiano.

Por ejemplo, en nuestra base de datos añadiremos un artista para el que no haya ninguna canción:

```
sqlite> INSERT INTO artista (nombre) VALUES("The Rolling
Stones");
sqlite>
```

La salida del producto cartesiano es la esperada:

```
sqlite> SELECT * FROM artista,cancion;
idartista|nombre|idcancion|titulo|idartista
1|Mike Oldfield|1|Tubular Bells|1
1|Mike Oldfield|2|Yellow Submarine|2
1|Mike Oldfield|3|Get Back|2
1|Mike Oldfield|4|Yesterday|2
1|Mike Oldfield|5|Moonlight Shadow|1
2|The Beatles|1|Tubular Bells|1
2|The Beatles|2|Yellow Submarine|2
2|The Beatles|3|Get Back|2
```

```
2|The Beatles|4|Yesterday|2
2|The Beatles|5|Moonlight Shadow|1
3|The Rolling Stones|1|Tubular Bells|1
3|The Rolling Stones|2|Yellow Submarine|2
3|The Rolling Stones|3|Get Back|2
3|The Rolling Stones|4|Yesterday|2
3|The Rolling Stones|5|Moonlight Shadow|1
sqlite>
```

Si se usa *LEFT JOIN* o *LEFT OUTER JOIN* con la cláusula correspondiente *ON* o *USING*, se generará una fila de salida para cada fila de la tabla de entrada de la izquierda a la que no corresponda ninguna fila de la tabla de la derecha. Las columnas correspondientes a la tabla de la derecha toman el valor NULL:

```
sqlite> SELECT * FROM artista LEFT JOIN cancion
USING(idartista);
idartista|nombre|idcancion|titulo
1|Mike Oldfield|5|Moonlight Shadow
1|Mike Oldfield|1|Tubular Bells
2|The Beatles|3|Get Back
2|The Beatles|2|Yellow Submarine
2|The Beatles|4|Yesterday
3|The Rolling Stones|NULL|NULL
sqlite>
```

Tambien podemos usar *LEFT* con composiciones naturales:

```
sqlite> SELECT * FROM artista NATURAL LEFT JOIN cancion;
idartista|nombre|idcancion|titulo
1|Mike Oldfield|5|Moonlight Shadow
1|Mike Oldfield|1|Tubular Bells
2|The Beatles|3|Get Back
2|The Beatles|2|Yellow Submarine
2|The Beatles|4|Yesterday
3|The Rolling Stones|NULL|NULL
sqlite>
```

SQLite no soporta composiciones externas a la derecha *RIGHT JOIN* o *RIGHT OUTER JOIN*.

8 Índices

Ya hemos visto algo sobre índices, cuando hablamos de las restricciones *PRIMARY KEY* y *UNIQUE*. Ahora veremos los índices en mayor detalle.

Los índices pueden ser creados en el momento de crear la tabla, ya sea mediante restricciones de columna:

```
sqlite> CREATE TABLE prueba(
   ...> clave INTEGER PRIMARY KEY,
   ...> texto TEXT);
sqlite> CREATE TABLE prueba2(
   ...> clave INTEGER CONTRAINT primaria PRIMARY KEY,
   ...> texto TEXT);
sqlite>
```

O mediante restricciones de tabla:

```
sqlite> CREATE TABLE prueba3(
   ...> clave INTEGER,
   ...> texto TEXT,
   ...> PRIMARY KEY (clave));
sqlite> CREATE TABLE prueba4(
   ...> clave INTEGER,
   ...> texto TEXT,
   ...> CONSTRAINT primaria PRIMARY KEY (clave));
sqlite>
```

La ventaja de este segundo método es que permite crear claves primaras sobre varias columnas:

```
sqlite> CREATE TABLE prueba5(
   ...> clave1 INTEGER,
   ...> clave2 INTEGER,
   ...> texto TEXT,
```

```
   ...> PRIMARY KEY (clave1,clave2));
sqlite>
```

La forma *CONSTRAINT <identificador> ...* permite crear identificadores para los índices asociados a la restricción. Esos identificadores se suelen usar para modificar (o alterar) las restricciones o para eliminarlas. En SQLite no les he encontrado ninguna utilidad.

Las mismas formas se pueden usar para crear claves únicas:

```
sqlite> CREATE TABLE prueba6(
   ...> clave INTEGER,
   ...> texto TEXT,
   ...> UNIQUE (clave));
sqlite>
```

Crear índices

También podemos crear índices sobre tablas existentes, o eliminarlos. Para ello disponemos de las sentencias CREATE INDEX y DROP INDEX.

```
sqlite> CREATE INDEX orden1 ON prueba3 (texto DESC);
sqlite> CREATE INDEX orden2 ON prueba4 (texto ASC,clave
DESC);
sqlite> CREATE UNIQUE INDEX orden3 ON prueba5 (clave1 DESC);
sqlite>
```

Podemos usar estos índices para obtener consultas ordenadas de forma más eficiente. Si existe un índice para un orden determinado, SQLite usará ese índice, evitando el gasto de tiempo y recursos de hacer una ordenación en el momento en que se necesita.

Borrar índices

Eliminar estos índices es muy sencillo:

```
sqlite> DROP INDEX orden1;
sqlite>
```

9 Transacciones

Las transacciones son operaciones sobre base de datos que se realizan de forma atómica en una o varias tablas. Pueden ser ejecutadas sobre la base de datos o deshechas en conjunto.

SQlite está optimizado para transacciones, es decir, las operaciones de inserción, modificación y borrado sobre tablas se ejecutarás más rápidamente si se realizan dentro de una transacción que si se realizan fuera.

Una transacción empieza con la sentencia BEGIN TRANSACTION, y termina con COMMIT TRANSACTION o con ROLLBACK TRANSACTION.

La palabra *TRANSACTION* es opcional y las sentencias COMMIT TRANSACTION y END TRANSACTION son equivalentes, por lo que podemos usar formas más reducidas, *BEGIN*, *END* o *COMMIT* y *ROLLBACK*.

En SQLite todas las modificaciones de base de datos se hacen dentro de una transacción, esto significa que cualquier sentencia, excepto SELECT, que no esté incluida en una transacción explícita, creará una transacción implícita que terminará justo después de ejecutada la sentencia.

Las transacciones se pueden anidar, estableciendo puntos de seguridad, mediante la sentencia SAVEPOINT. En el interior de una transacción podemos establecer tantos puntos de seguridad como necesitemos, cada uno con un nombre asociado. Las sentencias RELEASE y ROLLBACK se pueden aplicar a un punto de seguridad concreto.

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO cancion (titulo,idartista) VALUES("La
Macarena", 3);
sqlite> ROLLBACK;
```

```
sqlite> SELECT * FROM cancion;
idcancion|titulo|idartista
1|Tubular Bells|1
2|Yellow Submarine|2
3|Get Back|2
4|Yesterday|2
5|Moonlight Shadow|1
sqlite>
```

Al hacer un *rollback* de la transacción, la fila añadida no se transfiere a la base de datos, y los cambios realizados dentro de la transacción quedan sin efecto.

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO cancion (titulo,idartista)
VALUES("Sympathy for the Devil", 3);
sqlite> COMMIT;
sqlite> SELECT * FROM cancion;
idcancion|titulo|idartista
1|Tubular Bells|1
2|Yellow Submarine|2
3|Get Back|2
4|Yesterday|2
5|Moonlight Shadow|1
6|Sympathy for the Devil|3
sqlite>
```

Al terminar la transacción con un *commit*, los cambios se transfieren a la base de datos de forma definitiva.

Puntos de seguridad

Los puntos de seguridad nos permiten establecer puntos de retorno para hacer *rollback* o *commit* de forma anidada. En ocasiones ciertas transacciones dependen del resultado de operaciones posteriores, y lo que en un momento puede ser válido, más tarde puede quedar anulado.

```
sqlite> CREATE TABLE ejemplo (
...> id INTEGER,
...> nombre TEXT);
sqlite> INSERT INTO ejemplo VALUES(1,"Juan");
sqlite> SAVEPOINT punto1;
sqlite> INSERT INTO ejemplo VALUES(2,"Antonio");
sqlite> SAVEPOINT punto2;
sqlite> INSERT INTO ejemplo VALUES(3,"Pedro");
sqlite> ROLLBACK TO punto2;
sqlite> INSERT INTO ejemplo VALUES(4,"Carlos");
sqlite> RELEASE punto1;
sqlite> SELECT * FROM ejemplo;
id,nombre
1,Juan
2,Antonio
4,Carlos
sqlite>
```

En este ejemplo, la inserción de la fila (3,"Pedro") queda anulada al hacer un *rollback* del punto de seguridad "punto2". Sin embargo, el resto de las inserciones se ejecutan al hacer un *release* del "punto1".

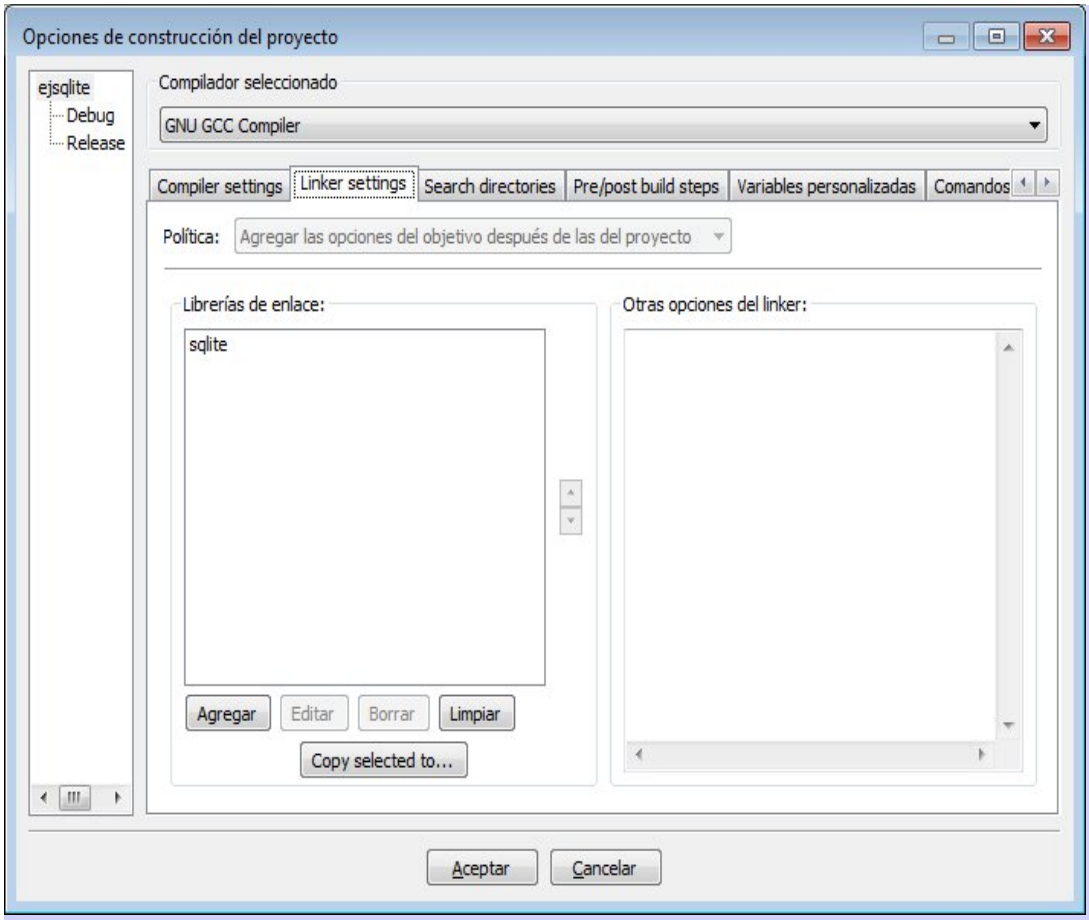
10 Preparación

Dado que SQLite no necesita un servidor de base de datos para funcionar, el uso de bases de datos en nuestras aplicaciones se simplifica enormemente. De hecho, no hay que realizar ninguna operación especial para usar bases de datos SQLite en un programa C/C++.

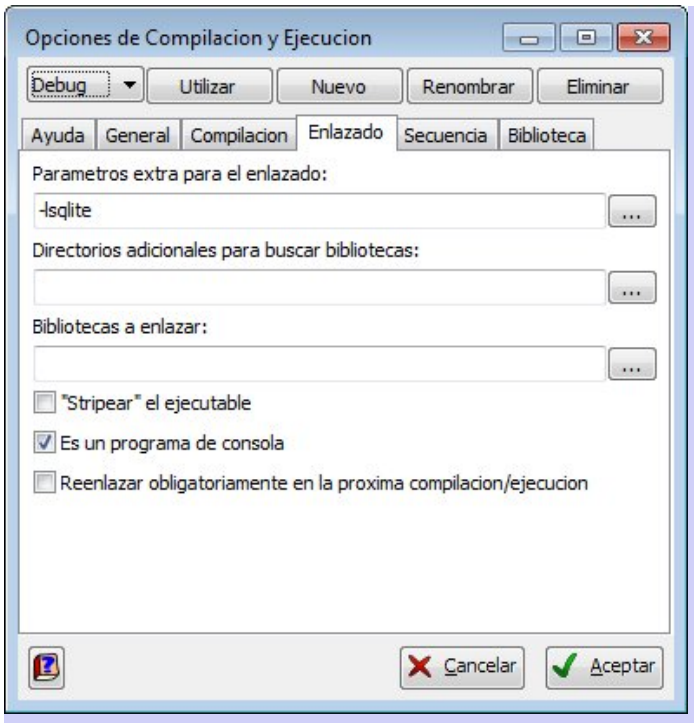
Sólo hay que añadir dos detalles para poder usar bases de datos en un programa. La primera, incluir el fichero de cabecera de SQLite en el código fuente:

```
#include <sqlite/sqlite3.h>
```

La segunda, hacer que el enlazador incluya la librería estática de SQLite. En cada compilador, y dependiendo de si usamos un IDE o no, esto se hará de formas diferentes. En nuestros ejemplos, usando Code::Blocks o ZinjaI, basta con añadir la opción "sqlite" en las opciones de construcción del proyecto:



Diálogo de opciones de construcción de Code::Blocks



Diálogo de opciones de construcción de Zinjal

Abrir una base de datos

Bien, lo primero es abrir la base de datos que vamos a usar en nuestro programa. Para ello usaremos la función del API sqlite3_open, que require dos parámetros. El primero es el nombre de fichero que contiene la base de datos, en nuestro caso "agenda.db". El segundo es un puntero doble a un objeto sqlite3.

El segundo parámetro es en realidad un valor de retorno de la función sqlite3_open. En nuestros programas usaremos un puntero a un objeto sqlite3 para almacenar una conexión con la base de datos.

Nota:

Muchas de las funciones del API de SQLite disponen de versiones para cadenas en formato UTF-8 y UTF-16. Este es el caso de sqlite3_open. Mientras sólo usemos caracteres ASCII para los nombres de fichero, esto no debería ser un problema.

El valor de retorno de sqlite3_open es un código de error, que indicará si la función ha tenido éxito, retornando SQLITE_OK, o si ha fracasado, con otro código que indica el error.

Este el el código para abrir la base de datos "agenda.db":

```
int rc;
sqlite3 *db;

rc = sqlite3_open("agenda.db", &db);
if(SQLITE_OK != rc) {
    cout << "Error: No se puede abrir la base de datos" << endl;
            return 1;
}
    ...
    sqlite3_close(db);
```

Por supuesto, una vez hayamos terminado de trabajar con la base de datos, deberemos cerrar la conexión con la base de datos. Para ello usaremos la función sqlite3_close.

11 Ejecutar sentencias

Las sentencias SQL son cadenas de texto, y por lo tanto, las almacenaremos en cadenas.

Por ejemplo:

```
SELECT * FROM usuario;
INSERT INTO usuario VALUES("Tulana", "tulana@dominio.com");
DROP usuario;
```

No podemos ejecutar las sentencias SQL directamente en nuestros programas. Cada sentencia requiere un proceso, que tendrá diferentes pasos, dependiendo del tipo de sentencia y de cómo decidamos trabajar con ella.

1. El primer paso, es compilar la sentencia. Para ello disponemos de la familia de funciones sqlite3_prepare, aunque es recomendable usar las que terminan en v2, dependiendo de si usamos codificación UTF-8 o UTF-16, sqlite3_prepare_v2 o sqlite3_prepare16_v2, respectivamente. El resto de las funciones se mantienen por compatibilidad, y no deben usarse.
   El primer argumento para estas funciones siempre es un puntero a un objeto de conexión de base de datos válido.
   El segundo es una cadena con la consulta SQL a compilar. La cadena puede contener varias sentencias, separadas con punto y coma.
   El tercer parámetro es la longitud máxima de la cadena que contiene la sentencia, o -1 si la el final de la sentencia está marcado con un carácter nulo.
   Para cada sentencia compilada se necesita mantener un objeto de tipo sqlite3_stmt. Las funciones sqlite3_prepare devuelven inicializado el puntero al objeto referenciado por el cuarto

argumento.
El quinto parámetro se usa cuando la cadena contiene más de una sentencia, y lo veremos más adelante.

2. Si la sentencia usa plantillas, el siguiente paso consiste en asignar valores a los parámetros. Si no usa plantillas este paso se omite. Estas asignaciones se hacen usando la familia de funciones sqlite3_bind*.
3. A continuación se ejecuta la sentencia, usando la función sqlite3_step.
   Esta función sólo necesita un parámetro, un puntero a un objeto sqlite3_stmt que contiene los datos de una sentencia compilada.
4. Opcionalmente podemos resetear la sentencia, y volver al punto 2. Esto lo haremos si los valores de la plantilla son distintos, por ejemplo, en el caso de sentencias INSERT.
5. O podemos mantener los parámetros y volver al punto 3, por ejemplo, para obtener la siguiente fila de una sentencia SELECT.
6. Finalmente, borramos la estructura asociada a la sentencia compilada, usando sqlite3_finalize.
   Esta función también necesita un único parámetro, un puntero a un puntero a un objeto sqlite3_stmt. Esta función libera el objeto asociado a una sentencia compilada.

Veamos diferentes ejemplos.

Sentencias que sólo se ejecutan una vez

En sentencias que sólo se ejecutarán una vez a lo largo del programa no necesitaremos crear bucles, ni almacenar la sentencia compilada una vez ejecutada. Usaremos una versión compacta del proceso. Por ejemplo, añadamos un índice a la tabla de usuarios:

```
CREATE INDEX nombre ON usuario (nombre);
```

```
    sqlite3_stmt *ppStmt;
    char consulta[64];
        ...

    strcpy(consulta, "CREATE INDEX nombre ON usuario
(nombre);");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        if(SQLITE_DONE != sqlite3_step(ppStmt)) {
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        sqlite3_finalize(ppStmt);
    }
```

Como la sentencia sólo necesita un paso, ejecutaremos la función sqlite3_step una vez. El valor de retorno de la función será SQLITE_DONE si se ha ejecutado correctamente.

Sentencias con varias salidas, sin parámetros

En sentencias que generan varios resultados y para las que no se necesita especificar parámetros usaremos una variante del proceso anterior:

```
    sqlite3_stmt *ppStmt;
    char consulta[64];
        ...

    strcpy(consulta, "SELECT rowid,nombre,email FROM
usuario;");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
```

```
    } else {
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "ID:     " << sqlite3_column_int(ppStmt,
0) << endl;
            cout << "Nombre: " <<
sqlite3_column_text(ppStmt, 1) << endl;
            cout << "email:  " <<
sqlite3_column_text(ppStmt, 2) << endl;
        }
        sqlite3_finalize(ppStmt);
    }
```

En este caso, sentencia puede necesitar cero o más pasos, ejecutaremos la función sqlite3_step mientras el valor de retorno sea SQLITE_ROW, que indica que se ha leído una fila.

Usaremos las funciones de la familia sqlite3_column_* para leer el valor de cada columna de la fila leída. El primero parámetro es siempre un puntero a un objeto sqlite3_stmt con los datos de la sentencia actual. El segundo es el índice de la columna, empezando en cero.

Hay que usar la función adecuada dependiendo del tipo de la columna que estemos leyendo.

Sentencias con parámetros

Podemos usar la misma sentencia compilada con distintos valores para los literales que contenga, de modo que nos sirva en distintas circunstancias. Dado que compilar una sentencia requiere más recursos y tiempo que modificar sus parámetros, es más eficiente compilar una única vez la sentencia.

Hay varias formas de indicar los parámetros SQL, podemos optar por la que nos interese en cada ocasión.

- Identificados mediante enteros. Hay dos opciones:
    - Una interrogación indica la posición de cada parámetro. Para hacer referencia a cada uno, se numeran de izquierda a derecha, empezando en 1:

```
SELECT * FROM usuario WHERE nombre=? AND email=?;
```

En este caso, el parámetro del nombre será en número 1 y el de email el número 2.

- Una interrogación, seguida de un valor entero indica la posición de cada parámetro. Ahora los números identificadores de cada parámetro serán los indicados:

```
SELECT * FROM usuario WHERE nombre=?2 AND email=?4;
```

En este caso, el parámetro del nombre será en número 2 y el de email el número 4.

- Identificados mediante literales. Los identifiadores literales empiezan con ":", "@" o "$", seguido de un literal alfabético:

```
SELECT * FROM usuario WHERE nombre=:nombre AND
email=:email;
```

Para conseguir el número de cada parámetro usaremos la función sqlite3_bind_parameter_index.

```
sqlite3_bind_text(ppStmt,
sqlite3_bind_parameter_index(ppStmt, ":nombre"),
"Fulano");
```

Nota: cuando se usan parámetros de tipo texto no se deben añadir las comillas delimitadoras, las funciones **sqlite3_bind*** insertan los parámetros adecuadamente, sean de texto o numéricos.

Veamos un ejemplo:

```
sqlite3_stmt *ppStmt;
char consulta[64];
...

strcpy(consulta, "SELECT numero FROM telefono WHERE
```

```
idusuario=@uid;");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        sqlite3_bind_int(ppStmt,
sqlite3_bind_parameter_index(ppStmt, "@uid"), 1);
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "telefono: " <<
sqlite3_column_text(ppStmt, 0) << endl;
        }
        sqlite3_finalize(ppStmt);
    }
```

Reutilizar sentencias compiladas

Si vamos a usar la misma sentencia, con los mismos o diferentes parámetros, bastará con compilarla una vez y guardar el objeto sqlite3_stmt asociado a ella. Cada vez que tengamos que ejecutar la sentencia desde el principio habrá que asignar los parámetros adecuados, si los hay, y resetaer la sentencia. Para resetear una sentencia compilada se usa la función sqlite3_reset.

Un ejemplo:

```
    sqlite3_stmt *ppStmt;
    sqlite3_stmt *ppStmt2;
    char consulta[128];
        ...

    strcpy(consulta, "SELECT rowid,nombre,email FROM
usuario;");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        sqlite3_close(db);
        return 0;
    }
    strcpy(consulta, "SELECT numero FROM telefono WHERE
idusuario=@uid;");
```

```
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt2,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(ppStmt);
        sqlite3_close(db);
        return 0;
    }
    while(SQLITE_ROW == sqlite3_step(ppStmt)) {
        cout << "ID:     " << sqlite3_column_int(ppStmt, 0)
<< endl;
        cout << "Nombre: " << sqlite3_column_text(ppStmt, 1)
<< endl;
        cout << "email:  " << sqlite3_column_text(ppStmt, 2)
<< endl;
        sqlite3_bind_int(ppStmt2,
sqlite3_bind_parameter_index(ppStmt2, "@uid"),
sqlite3_column_int(ppStmt, 0));
        while(SQLITE_ROW == sqlite3_step(ppStmt2)) {
            cout << " - telefono: " <<
sqlite3_column_text(ppStmt2, 0) << endl;
        }
        sqlite3_reset(ppStmt2);
    }
    sqlite3_finalize(ppStmt);
    sqlite3_finalize(ppStmt2);
```

Ejecución compacta de sentencias

Hay una alternativa a todo lo anterior, que podremos utilizar en ciertas circunstancias. Se trata de la función sqlite3_exec que es un envoltorio para las funciones sqlite3_prepare, sqlite3_step y sqlite3_finalize.

Esta función necesita cinco parámetros:

1. Una conexión de base de datos, es decir, un puntero válido a un objeto sqlite3.
2. Una cadena que contenga una o más sentencias SQL sin parámetros, separadas con punto y coma.
3. Un puntero a una función. Esta función es una retrollamada, que será invocada para cada fila resultante de evaluar la

sentencia SQL.
4. Un puntero void que será usado como primer parámetro para la retrollamada. Podemos usarlo para pasar parámetros personalizados a la retrollamada.
5. Un puntero a una cadena que se usará para almacenar un mensaje de error.

Cuando se trate de sentencias simples, que no proporcionen salidas, la sintaxis se simplifica mucho, ya que podemos omitir los tres últimos parámetros, por ejemplo:

```
    sqlite3_exec(db, "BEGIN", 0, 0, 0);
...
    sqlite3_exec(db, "END", 0, 0, 0);
```

Estas dos llamadas se pueden usar para empezar y terminar una transacción.

También se pueden usar esta función para generar listados, o iniciar listas de selección o menús en programas GUI, etc. Para eso tendremos que hacer uso de la función de retrollamada.

Esta función tiene este prototipo:

```
int callback(void *a_param, int argc, char **argv, char
**column);
```

Donde el primer parámetro es el que indicamos como cuarto en sqlite3_exec, el segundo es un contador, que indica el número de elementos en el tercero, que es un puntero a una lista de punteros char, que contendrán los valores de las columnas en la fila retornada por la iteración actual de la sentencia. El cuarto parámetro es otra lista de cadenas con los nombres de las columnas, y por lo tanto, con el mismo número de elementos que argv.

El primer parámetro podemos usarlo para lo que queramos, depende de cada caso. Si no lo necesitamos, usaremos el valor 0.

Veamos un ejemplo:

```
int lUsuario(void *a_param, int argc, char **argv, char
**column);
int lTelefono(void *a_param, int argc, char **argv, char
**column);
...
    char msg[512];
        int rc;

    rc = sqlite3_open("agenda.db", &db);
    if(SQLITE_OK != rc) {
        cout << "Error: No se puede abrir la base de datos"
<< endl;
        sqlite3_close(db);
        return 1;
    }

    sqlite3_exec(db, "SELECT rowid AS ID,nombre,email FROM
usuario ORDER BY nombre;", lUsuario, db, (char**)&msg);
    sqlite3_close(db);
...

int lUsuario(void *a_param, int argc, char **argv, char
**column) {
    char consulta[256];
    char msg[512];
    sqlite3 *db = (sqlite3*)a_param;

    for (int i=0; i < argc; i++)
        cout << column[i] << " : " << argv[i] << endl;
    sprintf(consulta, "SELECT numero FROM telefono WHERE
idusuario=\"%s\";", argv[0]);
    sqlite3_exec(db, consulta, lTelefono, 0, (char**)&msg);

    return 0;
}

int lTelefono(void *a_param, int argc, char **argv, char
**column) {
    for (int i=0; i < argc; i++)
        cout << column[i] << " : " << argv[i] << endl;
    return 0;
}
```

En la primera llamada a sqlite3_exec hemos usado el cuarto parámetro para pasar una copia del puntero a la conexión de base de datos a la función de retrollamada. De ese modo, dentro de esa función, podremos hacer otra consulta para obtener los números de teléfono.

12 Por qué usar transacciones

SQLite está optimizado para transacciones.

Para evitar errores de consistencia de bases de datos, todas las sentencias SQL son atómicas, es decir, en caso de interrupción del programa, por razones internas o externas, cualquier sentencia SQL o bien se ejecuta completamente, o no se ejecuta en absoluto. No hay término medio.

Esto significa que cualquier sentencia SQL que implique escribir en disco se asegurará de que la información realmente se escriba en el fichero, lo que implica más tiempo de ejecución.

Esto resulta muy evidente cuando se hacen muchas modificaciones, por ejemplo, si una rutina añade cientos o miles de filas en una tabla, hacer las inserciones una a una requiere sensiblemente más tiempo que hacerlo dentro de una transacción.

Este ejemplo ilustra la diferencia entre insertar 100 registros sin usar transacciones e inserter 10000 en una transacción:

```
// Agregar sqlite a opciones de enlazado
#include <sqlite/sqlite3.h>
#include <iostream>
#include <cstring>
#include <ctime>
#include <cstdio>

using namespace std;

int main() {
    int rc;
    sqlite3 *db;
    sqlite3_stmt *ppStmt;
    char consulta[128];
    char nombre[80];
    char email[60];
    time_t ti, tf;
```

```
    ti = time(0);
    strcpy(consulta, "INSERT INTO usuario VALUES(?,?)");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        for(int i = 0; i < 100; i++) {
            cout << i << (char)13;
            sprintf(nombre, "Nombre %4d", i);
            sprintf(email, "eMail %4d", i);
            sqlite3_bind_text(ppStmt, 1, nombre, -1,
SQLITE_STATIC);
            sqlite3_bind_text(ppStmt, 2, email, -1,
SQLITE_STATIC);
            if(SQLITE_DONE != sqlite3_step(ppStmt)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
            sqlite3_reset(ppStmt);
        }
        sqlite3_finalize(ppStmt);
    }
    tf = time(0);
    cout << "Tiempo para 100 sin transaccion: " << tf - ti
<< " segundos" << endl;

    sqlite3_exec(db, "DELETE FROM usuario WHERE nombre LIKE
\"Nombre %\";VACUUM;", 0, 0, 0);

    sqlite3_exec(db, "BEGIN", 0, 0, 0);
    ti = time(0);
    strcpy(consulta, "INSERT INTO usuario VALUES(?,?)");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        for(int i = 0; i < 10000; i++) {
            cout << i << (char)13;
            sprintf(nombre, "Nombre %4d", i);
            sprintf(email, "eMail %4d", i);
            sqlite3_bind_text(ppStmt, 1, nombre, -1,
SQLITE_STATIC);
            sqlite3_bind_text(ppStmt, 2, email, -1,
SQLITE_STATIC);
            if(SQLITE_DONE != sqlite3_step(ppStmt)) {
```

```
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
            sqlite3_reset(ppStmt);
        }
        sqlite3_finalize(ppStmt);
    }
    tf = time(0);
    cout << "Tiempo para 10000 registros con transaccion: "
<< tf - ti << " segundos" << endl;
    sqlite3_exec(db, "END", 0, 0, 0);

    sqlite3_exec(db, "DELETE FROM usuario WHERE nombre LIKE
\"Nombre %\";VACUUM;", 0, 0, 0);

    sqlite3_close(db);

    return 0;
}
```

La salida es elocuente (en mi ordenador al menos):

```
Tiempo para 100 sin transaccion: 15 segundos
Tiempo para 10000 registros con transaccion: 3 segundos
```

13 Ejemplo de aplicación SQLite

Vamos a desarrollar un programa basado en ejemplo 2 del curso de MySQL.

Recordemos el enunciado del problema:

> Nuestro segundo ejemplo es más complicado. Se trata de gestionar una biblioteca, y nuestro cliente quiere tener ciertas herramientas a su disposición para controlar libros, socios y préstamos. Adicionalmente se necesita un control de los ejemplares de cada libro, su ubicación y su estado, con vistas a su retirada o restitución, para esto último necesita información sobre editoriales a las que se deben pedir los libros.
>
> Tanto los libros como los socios estarán sujetos a ciertas categorías, de modo que según ellas cada libro podrá ser o no prestado a cada socio. Por ejemplo, si las categorías de los libros van de A a F, y la de los socios de B a F, un libro de categoría A nunca puede ser prestado a ningún socio. Estos libros sólo se pueden consultar en la biblioteca, pero no pueden salir de ella. Un libro de categoría B sólo a socios de categoría B, un libro de categoría C se podrá prestar a socios de categorías B y C, etc. Los libros de categoría F siempre pueden prestarse.
>
> El sistema debe proporcionar también un método de búsqueda para libros por parte de los socios, por tema, autor o título. El socio sólo recibirá información

sobre los libros de los que existen ejemplares, y sobre la categoría.

Además, se debe conservar un archivo histórico de préstamos, con las fechas de préstamo y devolución, así como una nota que el responsable de la biblioteca quiera hacer constar, por ejemplo, sobre el estado del ejemplar después de su devolución. Este archivo es una herramienta para la biblioteca que se puede usar para discriminar a socios "poco cuidadosos".

Los préstamos, generalmente, terminan con la devolución del libro, pero algunas veces el ejemplar se pierde o el plazo supera un periodo de tiempo establecido y se da por perdido. Estas circunstancias pueden cerrar un préstamo y provocan la baja del ejemplar (y en ocasiones la del socio :-). Nuestro archivo histórico debe contener información sobre si el libro fue devuelto o perdido.

Supondremos que vamos a desarrollar nuestro programa usando el compilador Code::Blocks, aunque el código y las acciones a tomar serán muy parecidas usando otros IDEs.

Empezaremos creando un nuevo proyecto, de tipo "Console application", en C++ y con el título "ejemplosqlite".

A continuación abrimos "Proyecto->Opciones para la construcción", vamos a la pestaña "linker settings", seleccionamos "ejemplosqlite", en el árbol de la izquierda, para que las opciones se apliquen tanto a la versión "debug" como a la "release", y agregamos la librería "sqlite".

Si la el fichero "sqlite3.dll" no está en una carpeta donde se pueda encontrar siempre (por ejemplo en system32, o cualquiera en el *path*), deberemos copiarla a la carpeta del proyecto. Haremos lo mismo con el fichero "sqlite3.exe", para poder consultar la base de datos desde consola, si fuera necesario.

Iremos añadiendo rutinas al programa, paso a paso, a medida que los vayamos explicando, de modo que cada fase de la aplicación pueda ser compilada y probada.

Crear la base de datos

Lo primero es crear la base de datos, si no existe, y abrirla si existe. Crear la base de datos es trivial, basta con intentar abrirla, si el fichero de base de datos no existe, se crea uno vacío. Nuestro primer programa es simple:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: main.cpp
 * fichero principal
 * Incluir en el enlazador la librería libsqlite.a (sqlite)
 * Facilitar el acceso a la librería de enlace dinámico
sqlite3.dll
 */

#include <iostream>
#include <sqlite/sqlite3.h>

using namespace std;

int main()
{
    int rc;
    sqlite3 *db;

    // Abrir base de datos
    rc = sqlite3_open("biblioteca.db", &db);
    if(SQLITE_OK != rc) {
        cout << "Error: No se puede abrir la base de datos"
<< endl;
                return 1;
    }
        // Aquí ira nuestro programa...
```

```
    // Cerrar base de datos
    sqlite3_close(db);
    return 0;
}
```

Después de ejecutar este programa podremos ver que se ha creado el fichero de la base de datos vacío.

Verificar si existen las tablas

El segundo paso es verificar si existen las tablas. Para ello haremos una consulta en cada una se ellas, y asumiremos que si se produce un error es porque la tabla no existe. SQLite no dispone de códigos de error específicos para indicar que una tabla no existe, aunque la función sqlite3_errmsg si detecta ese error.

Para nuestra segunda versión del programa crearemos una función que verifique la existencia de las nueve tablas que componen la base de datos, y que cree aquellas que no existen.

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: main.cpp
 * fichero principal
 * Incluir en el enlazador la librería libsqlite.a (sqlite)
 * Facilitar el acceso a la librería de enlace dinámico
sqlite3.dll
 */

#include <iostream>
#include <sqlite/sqlite3.h>

const int nTablas = 9;

bool VerificarTablas(sqlite3 *);

using namespace std;
```

```
int main()
{
    int rc;
    sqlite3 *db;

    // Abrir base de datos
    rc = sqlite3_open("biblioteca.db", &db);
    if(SQLITE_OK != rc) {
        cout << "Error: No se puede abrir la base de datos"
<< endl;
                return 1;
    }

    if(!VerificarTablas(db)) return -1;

    // Cerrar base de datos
    sqlite3_close(db);
    return 0;
}

bool VerificarTablas(sqlite3 *db) {
    char consulta[36];
    char *tabla[] = {
        "editorial",
        "libro",
        "autor",
        "tema",
        "ejemplar",
        "socio",
        "prestamo",
        "trata_sobre",
        "escrito_por"
    };
    char *tabla[] = {
        "editorial",
        "libro",
        "autor",
        "tema",
        "ejemplar",
        "socio",
        "prestamo",
        "trata_sobre",
        "escrito_por"
    };
    char *create[] = {
        "CREATE TABLE editorial("
          "claveeditorial INTEGER PRIMARY KEY,"
          "editorial TEXT,"
```

```
  "direccion TEXT,"
  "telefono TEXT);",
"CREATE TABLE libro("
  "clavelibro INTEGER PRIMARY KEY,"
  "titulo TEXT,"
  "idioma TEXT,"
  "formato TEXT,"
  "claveeditorial INTEGER "
  "REFERENCES editorial(claveeditorial) "
  "ON DELETE SET NULL "
  "ON UPDATE CASCADE);",
"CREATE TABLE autor("
  "claveautor INTEGER PRIMARY KEY,"
  "autor TEXT);",
"CREATE TABLE tema("
  "clavetema INTEGER PRIMARY KEY,"
  "tema TEXT);",
"CREATE TABLE ejemplar("
  "clavelibro INTEGER "
  "REFERENCES libro(clavelibro) "
  "ON DELETE CASCADE "
  "ON UPDATE CASCADE,"
  "numeroorden INTEGER NOT NULL,"
  "edicion INTEGER,"
  "ubicacion TEXT,"
  "categoria TEXT,"
  "PRIMARY KEY(clavelibro,numeroorden));",
"CREATE TABLE socio("
  "clavesocio INTEGER PRIMARY KEY,"
  "socio TEXT,"
  "direccion TEXT,"
  "telefono TEXT,"
  "categoria TEXT);",
"CREATE TABLE prestamo("
  "clavesocio INTEGER "
  "REFERENCES socio(clavesocio) "
  "ON DELETE SET NULL "
  "ON UPDATE CASCADE,"
  "clavelibro INTEGER "
  "REFERENCES ejemplar(clavelibro) "
  "ON DELETE SET NULL "
  "ON UPDATE CASCADE,"
  "numeroorden INTEGER,"
  "fecha_prestamo DATE NOT NULL,"
  "fecha_devolucion DATE DEFAULT NULL,"
  "notas TEXT);",
"CREATE TABLE trata_sobre("
  "clavelibro INTEGER NOT NULL "
```

```
            "REFERENCES libro(clavelibro) "
            "ON DELETE CASCADE "
            "ON UPDATE CASCADE,"
            "clavetema INTEGER NOT NULL "
            "REFERENCES tema(clavetema) "
            "ON DELETE CASCADE "
            "ON UPDATE CASCADE);",
          "CREATE TABLE escrito_por("
            "clavelibro INTEGER NOT NULL "
            "REFERENCES libro(clavelibro) "
            "ON DELETE CASCADE "
            "ON UPDATE CASCADE,"
            "claveautor INTEGER NOT NULL "
            "REFERENCES autor(claveautor) "
            "ON DELETE CASCADE "
            "ON UPDATE CASCADE);"
    };

    for(int i = 0; i < nTablas; i++) {
        sprintf(consulta, "SELECT COUNT(*) FROM %s;",
tabla[i]);
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "La tabla " << tabla[i] << " no existe."
<< endl;
            if(SQLITE_OK != sqlite3_exec(db, create[i], 0,
0, 0)) {
               cout << "Error al crear la tabla " <<
tabla[i] << endl;
               return false;
            }
        }
    }
    return true;
}
```

Menús de acceso

Nuestra aplicación estará escrita para consola, de modo que necesitaremos alguna forma de comunicarnos con ella. Por comodidad, para el usuario, interactuaremos mediante un menú.

Aprovecharemos la disponibilidad de la base de datos para almacenar los menús en una tabla. Lo normal sería no mezclar los

datos que maneja la aplicación con los que la hacen funcionar, pero en este caso nos limitaremos a añadir una tabla a la base de datos. Al contrario que el resto de las tablas, esta tendrá que contener información desde el principio, ya que los datos almacenados en esa tabla son imprescindibles para el funcionamiento de la aplicación.

Lo primero es diseñar un sistema de menús que nos permita hacer todas las tareas que indica el enunciado. Pero no nos interesa que todas las opciones se muestren desde el principio, ya que eso dificulta el manejo del programa. En lugar de eso crearemos una estructura en árbol para el menú, donde al activar ciertas opciones se despliegue un nuevo menú con las opciones disponibles. Por ejemplo, en un primer nivel mostraremos cinco opciones: "Ficheros maestros", "Libros", "Socios", "Préstamos" y "Salir". Si se elige "Ficheros maestros" se desplegará un segundo menú con las opciones de "Editoriales", "Autores", "Temas" y "Volver", etc.

Cada opción de menú tendrá asociado:

- Un identificador único de opción de menú.
- Un identificador de menú.
- Una letra que active la opción.
- El texto del menú.
- Un identificador de menú hijo.
- Un código de acción.

Aprovecharemos la estructura para almacenar los títulos de los submenús, para ello usaremos un valor especial de código de acción.

En nuestro proyecto añadiremos dos nuevos ficheros: menu.h y menu.cpp, que usaremos para declarar y definir las funciones, tipos y datos necesarios para manejar los menús.

En rigor, no necesitamos almacenar los menús en la base de datos, pero hacerlo tiene algunas ventajas, por ejemplo, permite traducir fácilmente los textos de los menús, inhibir algunas opciones, cambiar las teclas para activar ciertas opciones, etc.

El fichero menus.h queda así:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: menus.h
 * fichero de cabecera para implementar menús
 */

#include <sqlite/sqlite3.h>

#define TITULO              -1
#define SALIR               0
#define ABRIRMENU           1
#define NUEVAEDITORIAL      2
#define EDITAREDITORIAL     3
#define BORRAREDITORIAL     4
#define CONSULTAEDITORIAL   5
// Añadiremos más códigos de operación a medida que
implementemos nuevas opciones

struct stmenu { // Estructura para inicializar tabla de
menús
    int idmenu;
    char letra[2];
    char texto[64];
    int hijo;
    int accion;
};

// Prototipos:
bool IniciarMenu(sqlite3 *);
void MostrarMenu(sqlite3 *, int);
int LeerMenu(sqlite3 *, int&);
```

Usaremos tres funciones para manejar los menús:

IniciarMenu

Se encargará de crear la tabla "menu" y de insertar las filas necesarias. Si la tabla ya existe, no hará nada. Para obligar al programa a que genere la tabla de nuevo tendremos que

borrarla manualmente desde la consola "sqlite3", con la sentencia "DROP TABLE menu;".

MostrarMenu

Evidentemente, esta función se encargará de mostrar el menú adecuado, dependiendo del nivel, que pasaremos como segundo parámetro.

LeerMenu

Leerá la respuesta del usuario, y devolverá el código de acción asociado a la elección, y si eso implica un cambio de nivel, modificará el valor del segundo parámetro. Este parámetro se pasa por referencia, a la entrada indica el nivel actual, y a la salida el nuevo nivel.

La implementación de estas funciones se hace en "menu.cpp":

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: menus.cpp
 * fichero de implementación de menús
 */

#include <iostream>
#include "menus.h"

stmenu menu[] = {
    // Nivel, letra, texto, submenú, acción
    {1,"-","---MENU PRINCIPAL---",0,TITULO},
    {1, "1", "Maestros >", 2, ABRIRMENU},
    {1, "2", "Libros >", 3, ABRIRMENU},
    {1, "3", "Socios >", 4, ABRIRMENU},
    {1, "4", "Prestamos >", 5, ABRIRMENU},
    {1, "0", "Salir", 0, SALIR},
    {2,"-","---TABLAS MAESTRAS---",0,TITULO},
    {2, "1", "Editoriales >", 6, ABRIRMENU},
    {2, "2", "Autores >", 7, ABRIRMENU},
    {2, "3", "Temas >", 8, ABRIRMENU},
    {2, "0", "Salir <", 1, ABRIRMENU},
    {6,"-","---MENU EDITORIALES---",0,TITULO},
    {6, "1", "Nuevo", 0, NUEVAEDITORIAL},
```

```
    {6, "2", "Editar", 0, EDITAREDITORIAL},
    {6, "3", "Borrar", 0, BORRAREDITORIAL},
    {6, "4", "Consultar", 0, CONSULTAEDITORIAL},
    {6, "0", "Salir <", 2, ABRIRMENU},
    {3,"-","---MENU LIBROS---",0,TITULO},
    {3, "0", "Salir <", 1, ABRIRMENU},
    {4,"-","---MENU SOCIOS---",0,TITULO},
    {4, "0", "Salir <", 1, ABRIRMENU},
    {5,"-","---MENU PRESTAMOS---",0,TITULO},
    {5, "0", "Salir <", 1, ABRIRMENU},
    {7,"-","---MENU AUTORES---",0,TITULO},
    {7, "0", "Salir <", 2, ABRIRMENU},
    {8,"-","---MENU TEMAS---",0,TITULO},
    {8, "0", "Salir <", 2, ABRIRMENU}
};

using namespace std;

bool IniciarMenu(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[100];

    if(SQLITE_OK != sqlite3_exec(db, "SELECT COUNT(*) FROM
menu", 0, 0, 0)) {
        // if(SQLITE_OK != sqlite3_exec(db, "DROP TABLE
menu;", 0, 0, 0)) return false;
        if(SQLITE_OK != sqlite3_exec(db, "CREATE TABLE
menu(iditem INTEGER PRIMARY KEY,"
                                         "idmenu INTEGER,letra
TEXT,texto TEXT,hijo INTEGER,accion INTEGER);", 0, 0, 0))
return false;
        if(SQLITE_OK != sqlite3_exec(db, "BEGIN;", 0, 0, 0))
return false;

        strcpy(consulta, "INSERT INTO
menu(idmenu,letra,texto,hijo,accion)
VALUES(@mid,@let,@txt,@hij,@acc);");
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << sqlite3_errmsg(db) << endl;
            return false;
        } else {
            for(int i = 0; i < sizeof(menu)/sizeof(stmenu);
i++) {
                sqlite3_bind_int(ppStmt,
sqlite3_bind_parameter_index(ppStmt, "@mid"),
```

```
menu[i].idmenu);
                sqlite3_bind_text(ppStmt,
sqlite3_bind_parameter_index(ppStmt, "@let"), menu[i].letra,
-1, SQLITE_STATIC);
                sqlite3_bind_text(ppStmt,
sqlite3_bind_parameter_index(ppStmt, "@txt"), menu[i].texto,
-1, SQLITE_STATIC);
                sqlite3_bind_int(ppStmt,
sqlite3_bind_parameter_index(ppStmt, "@hij"), menu[i].hijo);
                sqlite3_bind_int(ppStmt,
sqlite3_bind_parameter_index(ppStmt, "@acc"),
menu[i].accion);
                sqlite3_step(ppStmt);
                sqlite3_reset(ppStmt);
            }
            sqlite3_finalize(ppStmt);
        }
        if(SQLITE_OK != sqlite3_exec(db, "COMMIT;", 0, 0,
0)) return false;
    }
    return true;
}

void MostrarMenu(sqlite3 *db, int nivel) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[80];
    int nLineas;
    char titulo[64];

    // Contar items:
    sprintf(consulta, "SELECT COUNT(*) FROM menu WHERE
idmenu=%d AND accion!=-1;", nivel);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc==SQLITE_OK ){
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            nLineas = sqlite3_column_int(ppStmt, 0);
        }
        sqlite3_finalize(ppStmt);
    }

    // 24 líneas en blanco (Borrar pantalla):
    for(int i = 0; i < 24; i++) cout << endl;

    // Titulo:
    strcpy(titulo, "---MENU---"); // Titulo por defecto
    sprintf(consulta, "SELECT texto FROM menu WHERE
```

```
idmenu=%d AND accion=-1;", nivel);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc==SQLITE_OK ){
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            strcpy(titulo, (const
char*)sqlite3_column_text(ppStmt, 0));
        }
        sqlite3_finalize(ppStmt);
    }

    cout << "\t\t" << titulo << "\n" << endl;

    strcpy(consulta, "SELECT letra,texto FROM menu WHERE
idmenu=@mid AND accion!=-1;");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        sqlite3_bind_int(ppStmt,
sqlite3_bind_parameter_index(ppStmt, "@mid"), nivel);
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << sqlite3_column_int(ppStmt, 0) << ") " <<
sqlite3_column_text(ppStmt, 1) << endl;
        }
        sqlite3_finalize(ppStmt);
    }
    // Más líneas en blanco:
    for(int i = 0; i < (18-nLineas); i++) cout << endl;
    cout << "\tOpcion: ";
}

int LeerMenu(sqlite3 *db, int& nivel) {
    char resp[2];
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[64];
    int retval;

    cin >> resp;
    sprintf(consulta, "SELECT hijo,accion FROM menu WHERE
letra='%s' AND idmenu=%d;", resp, nivel);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
```

```
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            retval = sqlite3_column_int(ppStmt, 1);
            if(retval == ABRIRMENU) nivel =
sqlite3_column_int(ppStmt, 0);
        }
        sqlite3_finalize(ppStmt);
    }
    return retval;
}
```

Por último, modificaremos el fichero "main.cpp" para integrar los menús:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: main.cpp
 * fichero principal
 * Incluir en el enlazador la librería libsqlite.a (sqlite)
 * Facilitar el acceso a la librería de enlace dinámico
sqlite3.dll
 */

#include <iostream>
#include <sqlite/sqlite3.h>
#include "menus.h"

const int nTablas = 9;

bool VerificarTablas(sqlite3 *);

using namespace std;

int main()
{
    int rc;
    sqlite3 *db;
    int nivel=1;
    bool salir;

    // Abrir base de datos
    rc = sqlite3_open("biblioteca.db", &db);
```

```
    if(SQLITE_OK != rc) {
        cout << "Error: No se puede abrir la base de datos"
<< endl;
                return 1;
    }

    if(!VerificarTablas(db)) return -1;
    if(!IniciarMenu(db)) return -1;

    do {
        MostrarMenu(db, nivel);
        switch(LeerMenu(db, nivel)) {
            case ABRIRMENU:
               // Nada que hacer.
                           break;
            case NUEVAEDITORIAL:
            case EDITAREDITORIAL:
            case BORRAREDITORIAL:
            case CONSULTAEDITORIAL:
               cout << "No implementado" << endl;
               break;
            case SALIR:
               salir = true;
        }
    } while(!salir);

    // Cerrar base de datos
    sqlite3_close(db);
    return 0;
}
...
// El resto es igual que antes...
```

Las modificaciones en main se limitan a añadir una llamada a IniciarMenu, y un bucle en el que se muestra el menú, se lee la respuesta del usuario y se procesa el resultado, hasta que se elija la opción de salir.

Ahora ya podemos empezar a implementar las distintas opciones. Usaremos dos ficheros para cada submenú, uno con los prototipos de las funciones y otro con la implementación.

Procesar tabla de editoriales

Ahora que tenemos divididas las tareas, podemos empezar a codificarlas una a una. Empezaremos con las editoriales, diseñando las funciones para añadir, editar, borrar y consultar datos.

Como siempre, añadiremos dos ficheros al proyecto, uno con los prototipos de funciones y otro con la implementación.

Fichero de cabecera:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: editorial.h
 * fichero de cabecera para manupular datos de editoriales
 */

#ifndef __EDITORIAL_H__
#define __EDITORIAL_H__

#include <sqlite/sqlite3.h>

void NuevaEditorial(sqlite3 *);
int ListaEditoriales(sqlite3 *);
void EditarEditorial(sqlite3 *);
void BorrarEditorial(sqlite3 *);
void BuscarEditorial(sqlite3 *);
#endif
```

El fichero de implementación es algo más largo:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: editorial.cpp
 * fichero de implementación para manupular datos de
editoriales
 */
```

```
#include <iostream>
#include <iomanip>
#include "editorial.h"

using namespace std;

void NuevaEditorial(sqlite3 *db) {
    char nombre[64];
    char direccion[128];
    char telefono[32];
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    bool existe, ignorar=false;
    char nombre2[64];
    int clave;
    char resp[2];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira el nombre, direccion y
telefono de la editorial." << endl;

    cin.ignore();
    cout << "Nombre: ";
    cin.getline(nombre, 64);
    cout << "Direccion: ";
    cin.getline(direccion,128);
    cout << "Telefono: ";
    cin.getline(telefono,32);

    // Verificar si el nombre existe ya:
    sprintf(consulta, "SELECT claveeditorial,editorial FROM
editorial WHERE editorial LIKE '%s';", nombre);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    existe=false;
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            existe = true;
            clave = sqlite3_column_int(ppStmt, 0);
            strncpy(nombre2, (const
char*)sqlite3_column_text(ppStmt, 1), 64);
            nombre2[63]=0;
        }
        sqlite3_finalize(ppStmt);
    }
```

```
    if(!existe) {
        sprintf(consulta, "INSERT INTO
editorial(editorial,direccion,telefono)
VALUES('%s','%s','%s');",
                nombre, direccion, telefono);
    } else {
        cout << "Ya existe una editorial con el nombre " <<
nombre2 << " (s)obrescribir, insert(a)r o (i)gnorar: " <<
endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                sprintf(consulta, "UPDATE editorial SET
editorial='%s',direccion='%s',telefono='%s' WHERE
claveeditorial=%d;",
                        nombre, direccion, telefono, clave);
                break;
            case 'a':
                sprintf(consulta, "INSERT INTO
editorial(editorial,direccion,telefono)
VALUES('%s','%s','%s');",
                        nombre, direccion, telefono);
                break;
            case 'i':
            default:
                ignorar=true;
        }
    }
    if(!ignorar) {
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Editorial insertada" << endl;
    }
    cin.ignore();
}

int ListaEditoriales(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
```

```
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir editorial" << endl << endl;
        sprintf(consulta, "SELECT claveeditorial,editorial
FROM editorial ORDER BY nombre LIMIT 20 OFFSET %d;",
desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
```

```
}

void EditarEditorial(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    char direccion[128];
    char telefono[32];
    int i;
    int fila;
    bool salir=true;

    fila = ListaEditoriales(db);

    // Editar:
    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT editorial,direccion,telefono
FROM editorial WHERE claveeditorial='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Direccion: " <<
sqlite3_column_text(ppStmt, 1) << endl;
            cout << "Telefono:  " <<
sqlite3_column_text(ppStmt, 2) << endl;
            cout << "Dejar en blanco los campos que no se
quieren modifiar" << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Nombre: ";
        cin.getline(nombre, 64);
        cout << "Direccion: ";
        cin.getline(direccion,128);
        cout << "Telefono: ";
        cin.getline(telefono,32);
        if(strlen(nombre)>0) {
            sprintf(consulta, "UPDATE editorial SET
```

```
editorial='%s' WHERE claveeditorial=%d;", nombre, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(direccion)>0) {
            sprintf(consulta, "UPDATE editorial SET
direccion='%s' WHERE claveeditorial=%d;", direccion, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(telefono)>0) {
            sprintf(consulta, "UPDATE editorial SET
telefono='%s' WHERE claveeditorial=%d;", telefono, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        cout << "Editorial modificada" << endl;
        cin.ignore();
    }
}

void BorrarEditorial(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int fila;
    char resp[2];
    int i;
    bool salir=true;

    fila = ListaEditoriales(db);
    cout << "Borrar: " << fila << endl;

    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT editorial,direccion,telefono
FROM editorial WHERE claveeditorial='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
```

```
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Direccion: " <<
sqlite3_column_text(ppStmt, 1) << endl;
            cout << "Telefono:  " <<
sqlite3_column_text(ppStmt, 2) << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Borrar este registro? (s/n)" << endl;
        cin >> resp;
        if(resp[0] == 's' || resp[0] == 'S') {
            sprintf(consulta, "DELETE FROM editorial WHERE
claveeditorial=%d;", fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
    }
    cout << "Editorial borrada" << endl;
    cin.ignore();
}

void BuscarEditorial(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    char direccion[128];
    char telefono[32];
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Introducir cadenas de busqueda, _ para comodin
de caracter, % para comodin de cadena" << endl;
    cout << "Dejar en blanco para ignorar el campo en la
busqueda" << endl;
```

```
    cin.ignore();
    cout << "Nombre: ";
    cin.getline(nombre, 64);
    cout << "Direccion: ";
    cin.getline(direccion,128);
    cout << "Telefono: ";
    cin.getline(telefono,32);

    if(strlen(nombre) == 0) strcpy(nombre, "%");
    if(strlen(direccion) == 0) strcpy(direccion, "%");
    if(strlen(telefono) == 0) strcpy(telefono, "%");
    sprintf(consulta, "SELECT editorial,direccion,telefono
FROM editorial "
            "WHERE editorial LIKE '%s' AND direccion LIKE
'%s' AND telefono LIKE '%s';",
            nombre, direccion, telefono);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout.setf(ios::left);
            cout.width(64);
            cout << sqlite3_column_text(ppStmt, 0) << endl;
            cout << sqlite3_column_text(ppStmt, 1) << " ";
            cout.width(12);
            cout << sqlite3_column_text(ppStmt, 2) << endl;
            i +=2;
            if(!(i % 22)) {
                cout << "Pulsa return";
                cin.ignore();
                cin.get();
            }
        }
        sqlite3_finalize(ppStmt);
    }
    cin.ignore();
    cin.get();
}
```

Por último, modificamos el fichero "main.cpp", añadiendo un include para el nuevo fichero de cabecera, y modificando el bucle de procesamiento de menú:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: main.cpp
 * fichero principal
 * Incluir en el enlazador la librería libsqlite.a (sqlite)
 * Facilitar el acceso a la librería de enlace dinámico
sqlite3.dll
 */

#include <iostream>
#include <sqlite/sqlite3.h>
#include "menus.h"
#include "editorial.h"

const int nTablas = 9;

bool VerificarTablas(sqlite3 *);

using namespace std;

int main()
{
    int rc;
    sqlite3 *db;
    int nivel=1;
    bool salir;

    // Abrir base de datos
    rc = sqlite3_open("biblioteca.db", &db);
    if(SQLITE_OK != rc) {
        cout << "Error: No se puede abrir la base de datos"
<< endl;
                return 1;
    }

    if(!VerificarTablas(db)) return -1;
    if(!IniciarMenu(db)) return -1;

    do {
        MostrarMenu(db, nivel);
        switch(LeerMenu(db, nivel)) {
            case ABRIRMENU:
               // Nada que hacer.
```

```
                break;
            case NUEVAEDITORIAL:
                NuevaEditorial(db);
                break;
            case EDITAREDITORIAL:
                EditarEditorial(db);
                break;
            case BORRAREDITORIAL:
                BorrarEditorial(db);
                break;
            case CONSULTAEDITORIAL:
                BuscarEditorial(db);
                break;
            case SALIR:
                salir = true;
        }
    } while(!salir);

    // Cerrar base de datos
    sqlite3_close(db);
    return 0;
}
...
```

Procesar tablas de autores y temas

Las funciones para el tratamiento de las tablas de *autores* y *temas* son similares, y se pueden adaptar de la de *editoriales* con muy pocos cambios.

Fichero 'autor.h':

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: autor.h
 * fichero de cabecera para manupular datos de autores
 */

#ifndef __AUTOR_H__
#define __AUTOR_H__
```

```
#include <sqlite/sqlite3.h>

void NuevoAutor(sqlite3 *);
int ListaAutores(sqlite3 *);
void EditarAutor(sqlite3 *);
void BorrarAutor(sqlite3 *);
void BuscarAutor(sqlite3 *);
#endif
```

Fichero 'tema.h':

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: tema.h
 * fichero de cabecera para manupular datos de temas
 */

#ifndef __TEMA_H__
#define __TEMA_H__

#include <sqlite/sqlite3.h>

void NuevoTema(sqlite3 *);
int ListaTemnas(sqlite3 *);
void EditarTema(sqlite3 *);
void BorrarTema(sqlite3 *);
void BuscarTema(sqlite3 *);
#endif
```

Fichero 'autor.cpp':

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: autor.cpp
 * fichero de implementación para manupular datos de autores
```

```
 */

#include <iostream>
#include <iomanip>
#include "autor.h"

using namespace std;

void NuevoAutor(sqlite3 *db) {
    char nombre[64];
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    bool existe, ignorar=false;
    char nombre2[64];
    int clave;
    char resp[2];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira el nombre, del autor."
<< endl;

    cin.ignore();
    cout << "Nombre: ";
    cin.getline(nombre, 64);

    // Verificar si el nombre existe ya:
    sprintf(consulta, "SELECT claveautor,autor FROM autor
WHERE autor LIKE '%s';", nombre);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    existe=false;
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            existe = true;
            clave = sqlite3_column_int(ppStmt, 0);
            strncpy(nombre2, (const
char*)sqlite3_column_text(ppStmt, 1), 64);
            nombre2[63]=0;
        }
        sqlite3_finalize(ppStmt);
    }

    if(!existe) {
        sprintf(consulta, "INSERT INTO autor(autor)
VALUES('%s');", nombre);
```

```
    } else {
        cout << "Ya existe un autor con el nombre " <<
nombre2 << " (s)obrescribir, insert(a)r o (i)gnorar: " <<
endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                sprintf(consulta, "UPDATE autor SET
autor='%s' WHERE claveautor=%d;", nombre,  clave);
                break;
            case 'a':
                sprintf(consulta, "INSERT INTO autor(autor)
VALUES('%s');", nombre);
                break;
            case 'i':
            default:
                ignorar=true;
        }
    }
    if(!ignorar) {
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Autor insertado" << endl;
    }
    cin.ignore();
}

int ListaAutores(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir autor" << endl << endl;
        sprintf(consulta, "SELECT claveautor,autor FROM
autor ORDER BY autor LIMIT 20 OFFSET %d;", desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
```

```
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
}

void EditarAutor(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    int i;
    int fila;
    bool salir=true;
```

```
    fila = ListaAutores(db);

    // Editar:
    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT autor FROM autor WHERE
claveautor='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Dejar en blanco los campos que no se
quieren modifiar" << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Nombre: ";
        cin.getline(nombre, 64);
        if(strlen(nombre)>0) {
            sprintf(consulta, "UPDATE autor SET autor='%s'
WHERE claveautor=%d;", nombre, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        cout << "Autor modificado" << endl;
        cin.ignore();
    }
}

void BorrarAutor(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int fila;
    char resp[2];
    int i;
    bool salir=true;
```

```
    fila = ListaAutores(db);
    cout << "Borrar: " << fila << endl;

    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT autor FROM autor WHERE
claveautor='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Borrar este registro? (s/n)" << endl;
        cin >> resp;
        if(resp[0] == 's' || resp[0] == 'S') {
            sprintf(consulta, "DELETE FROM autor WHERE
claveautor=%d;", fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
    }
    cout << "Autor borrado" << endl;
    cin.ignore();
}

void BuscarAutor(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Introducir cadenas de busqueda, _ para comodin
```

```
de caracter, % para comodin de cadena" << endl;
    cout << "Dejar en blanco para ignorar el campo en la
busqueda" << endl;
    cin.ignore();
    cout << "Nombre: ";
    cin.getline(nombre, 64);

    if(strlen(nombre) == 0) strcpy(nombre, "%");
    sprintf(consulta, "SELECT autor FROM autor WHERE autor
LIKE '%s';", nombre);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout.setf(ios::left);
            cout.width(64);
            cout << sqlite3_column_text(ppStmt, 0) << endl;
            i++;
            if(!(i % 22)) {
                cout << "Pulsa return";
                cin.ignore();
                cin.get();
            }
        }
        sqlite3_finalize(ppStmt);
    }
    cin.ignore();
    cin.get();
}
```

Fichero 'tema.cpp':

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: tema.cpp
 * fichero de implementación para manupular datos de temas
 */

#include <iostream>
```

```
#include <iomanip>
#include "tema.h"

using namespace std;

void NuevoTema(sqlite3 *db) {
    char nombre[64];
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    bool existe, ignorar=false;
    char nombre2[64];
    int clave;
    char resp[2];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira el nombre del tema."
<< endl;

    cin.ignore();
    cout << "Nombre: ";
    cin.getline(nombre, 64);

    // Verificar si el nombre existe ya:
    sprintf(consulta, "SELECT clavetema,tema FROM tema WHERE
tema LIKE '%s';", nombre);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    existe=false;
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            existe = true;
            clave = sqlite3_column_int(ppStmt, 0);
            strncpy(nombre2, (const
char*)sqlite3_column_text(ppStmt, 1), 64);
            nombre2[63]=0;
        }
        sqlite3_finalize(ppStmt);
    }

    if(!existe) {
        sprintf(consulta, "INSERT INTO tema(tema)
VALUES('%s');", nombre);
    } else {
        cout << "Ya existe un tema con el nombre " <<
nombre2 << " (s)obrescribir, insert(a)r o (i)gnorar: " <<
```

```
endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                sprintf(consulta, "UPDATE tema SET tema='%s'
WHERE clavetema=%d;", nombre,  clave);
                break;
            case 'a':
                sprintf(consulta, "INSERT INTO tema(tema)
VALUES('%s');", nombre);
                break;
            case 'i':
            default:
                ignorar=true;
        }
    }
    if(!ignorar) {
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Tema insertado" << endl;
    }
    cin.ignore();
}

int ListaTemas(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir tema" << endl << endl;
        sprintf(consulta, "SELECT clavetema,tema FROM tema
ORDER BY tema LIMIT 20 OFFSET %d;", desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
```

```
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
}

void EditarTema(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    int i;
    int fila;
    bool salir=true;

    fila = ListaTemas(db);

    // Editar:
```

```
    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT tema FROM tema WHERE
clavetema='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Dejar en blanco los campos que no se
quieren modifiar" << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Nombre: ";
        cin.getline(nombre, 64);
        if(strlen(nombre)>0) {
            sprintf(consulta, "UPDATE tema SET tema='%s'
WHERE clavetema=%d;", nombre, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        cout << "Tema modificado" << endl;
        cin.ignore();
    }
}

void BorrarTema(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int fila;
    char resp[2];
    int i;
    bool salir=true;

    fila = ListaTemas(db);
    cout << "Borrar: " << fila << endl;
```

```
    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT tema FROM tema WHERE
clavetema='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Borrar este registro? (s/n)" << endl;
        cin >> resp;
        if(resp[0] == 's' || resp[0] == 'S') {
            sprintf(consulta, "DELETE FROM tema WHERE
clavetema=%d;", fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
    }
    cout << "Tema borrado" << endl;
    cin.ignore();
}

void BuscarTema(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Introducir cadenas de busqueda, _ para comodin
de caracter, % para comodin de cadena" << endl;
    cout << "Dejar en blanco para ignorar el campo en la
busqueda" << endl;
```

```
    cin.ignore();
    cout << "Nombre: ";
    cin.getline(nombre, 64);

    if(strlen(nombre) == 0) strcpy(nombre, "%");
    sprintf(consulta, "SELECT tema FROM tema WHERE tema LIKE
'%s';", nombre);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout.setf(ios::left);
            cout.width(64);
            cout << sqlite3_column_text(ppStmt, 0) << endl;
            i++;
            if(!(i % 22)) {
                cout << "Pulsa return";
                cin.ignore();
                cin.get();
            }
        }
        sqlite3_finalize(ppStmt);
    }
    cin.ignore();
    cin.get();
}
```

En 'menus.h' añadimos la definición de las macros para las nuevas opciones de menú:

```
#define NUEVOAUTOR          6
#define EDITARAUTOR         7
#define BORRARAUTOR         8
#define CONSULTAAUTOR       9
#define NUEVOTEMA           10
#define EDITARTEMA          11
#define BORRARTEMA          12
#define CONSULTATEMA        13
```

En 'menus.cpp' añadimos los valores del array de menú para las nuevas opciones:

```
    {7,"-","---MENU AUTORES---",0,TITULO},
    {7, "1", "Nuevo", 0, NUEVOAUTOR},
    {7, "2", "Editar", 0, EDITARAUTOR},
    {7, "3", "Borrar", 0, BORRARAUTOR},
    {7, "4", "Consultar", 0, CONSULTAAUTOR},
    {7, "0", "Salir <", 2, ABRIRMENU},
    {8,"-","---MENU TEMAS---",0,TITULO},
    {8, "1", "Nuevo", 0, NUEVOTEMA},
    {8, "2", "Editar", 0, EDITARTEMA},
    {8, "3", "Borrar", 0, BORRARTEMA},
    {8, "4", "Consultar", 0, CONSULTATEMA},
    {8, "0", "Salir <", 2, ABRIRMENU}
```

Finalmente, modificamos el fichero 'main.cpp' para añadir los ficheros de cabecera y los 'case' para procesar las nuevas opciones:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Marzo de 2012
 * Fichero: main.cpp
 * fichero principal
 * Incluir en el enlazador la librería libsqlite.a (sqlite)
 * Facilitar el acceso a la librería de enlace dinámico
sqlite3.dll
 */

#include <iostream>
#include <sqlite/sqlite3.h>
#include "menus.h"
#include "editorial.h"
#include "autor.h"
#include "tema.h"

const int nTablas = 9;

bool VerificarTablas(sqlite3 *);
```

```
using namespace std;

int main()
{
    int rc;
    sqlite3 *db;
    int nivel=1;
    bool salir;

    // Abrir base de datos
    rc = sqlite3_open("biblioteca.db", &db);
    if(SQLITE_OK != rc) {
        cout << "Error: No se puede abrir la base de datos"
<< endl;
                return 1;
    }

    if(!VerificarTablas(db)) return -1;
    if(!IniciarMenu(db)) return -1;

    do {
        MostrarMenu(db, nivel);
        switch(LeerMenu(db, nivel)) {
            case ABRIRMENU:
               // Nada que hacer.
               break;
            case NUEVAEDITORIAL:
               NuevaEditorial(db);
               break;
            case EDITAREDITORIAL:
               EditarEditorial(db);
               break;
            case BORRAREDITORIAL:
               BorrarEditorial(db);
               break;
            case CONSULTAEDITORIAL:
               BuscarEditorial(db);
               break;
            case NUEVOAUTOR:
               NuevoAutor(db);
               break;
            case EDITARAUTOR:
               EditarAutor(db);
               break;
            case BORRARAUTOR:
               BorrarAutor(db);
               break;
            case CONSULTAAUTOR:
```

```
                BuscarAutor(db);
                break;
            case NUEVOTEMA:
                NuevoTema(db);
                break;
            case EDITARTEMA:
                EditarTema(db);
                break;
            case BORRARTEMA:
                BorrarTema(db);
                break;
            case CONSULTATEMA:
                BuscarTema(db);
                break;
            case SALIR:
                salir = true;
        }
    } while(!salir);

    // Cerrar base de datos
    sqlite3_close(db);
    return 0;
}
...
```

Procesar tabla de libros

Esta tabla tiene más trabajo que hacer, ya que la tabla de libros incluye referencias a autores, temas y editoriales. Además, deberemos manejar ejemplares.

Crear, modificar o borrar libros tiene muchas más implicaciones que hacerlo con las tablas que hemos visto hasta ahora. Lamentablemente, escribir un programa como este para consola, usando sólo funciones estántar y el API de SQLite, limita mucho la facilidad de edición en cuanto a la elección de valores válidos desde listas, etc.

En un programa más elaborado crearíamos tablas auxiliares para almacenar idiomas y formatos, de modo que podamos seleccionarlos desde una lista, y minimizar los errores y facilitar las

búsquedas. En este ejemplo dejaremos que el usuario escriba los valores de esos campos como quiera.

Por ello usaremos más opciones de menú. Por ejemplo, en lugar de seleccionar los autores desde una lista, añadiremos una opción de menú para añadir o eliminar autores a un libro.

Haremos lo mismo con los temas. Hay que tener en cuenta que un libro puede tener varios autores y tratar sobre varios temas.

Como en los casos anteriores, añadiremos los nuevos identificadores de menú en "menus.h":

```
...
#define NUEVOLIBRO          14
#define EDITARLIBRO         15
#define ANEXARAUTOR         16
#define ELIMINARAUTOR       17
#define ANEXARTEMA          18
#define ELIMINARTEMA        19
#define BORRARLIBRO         20
#define CONSULTALIBRO       21
...
```

También añadiremos las nuevas entradas en la tabla de menús. Para que las tenga en cuenta, hay que eliminar la tabla de menús, lo haremos desde la consola de SQLite, con la orden "DROP TABLE menu;".

```
    {3, "-", "---MENU LIBROS---",0,TITULO},
    {3, "1", "Nuevo", 0, NUEVOLIBRO},
    {3, "2", "Editar", 0, EDITARLIBRO},
    {3, "3", "Anexar autor a libro", 0, ANEXARAUTOR},
    {3, "4", "Eliminar autor de libro", 0, ELIMINARAUTOR},
    {3, "5", "Anexar tema a libro", 0, ANEXARTEMA},
    {3, "6", "Eliminar tema de libro", 0, ELIMINARTEMA},
    {3, "7", "Borrar", 0, BORRARLIBRO},
    {3, "8", "Consultar", 0, CONSULTALIBRO},
    {3, "9", "Ejemplares >", 9, ABRIRMENU},
    {3, "0", "Salir <", 1, ABRIRMENU},
...
```

```
    {9, "-", "---MENU EJEMPLARES---",0,TITULO},
    {9, "0", "Salir <", 3, ABRIRMENU}
```

En el fichero "main.cpp" añadiremos las nuevas opciones al bucle de tratamiento de menú:

```
            case NUEVOLIBRO:
               NuevoLibro(db);
               break;
            case EDITARLIBRO:
               EditarLibro(db);
               break;
            case ANEXARAUTOR:
               AnexarAutor(db);
               break;
            case ANEXARTEMA:
               AnexarTema(db);
               break;
            case ELIMINARAUTOR:
               EliminarAutorLibro(db);
               break;
            case ELIMINARTEMA:
               EliminarTemaLibro(db);
               break;
            case BORRARLIBRO:
               BorrarLibro(db);
               break;
            case CONSULTALIBRO:
               BuscarLibro(db);
               break;
```

Evitar repeticiones de autores y temas

Cuando diseñamos las tablas para este problema, no creamos una clave para las tablas "escrito_por" y "trata_sobre". Esto hace que sea posible que existan varios registros con los mismos valores de clavelibro/claveautor y clavelibro/clavetema. Es decir, que se especifique el mismo autor o tema varias veces para el mismo libro.

Podríamos hacer que el código del programa evitase esto, verificando si la fila ya existe antes de insertarla, pero es más

sencillo dejar la tarea al motor de base de datos, creando un índice único para cada tabla.

Para ello modificaremos las consultas SQL en la función 'VerificarTablas' de modo que se creen esas restricciones para las tablas "escrito_por" y "trata_sobre". De modo que para que se creen las tablas con las nuevas características, eliminaremos las tablas manualmente.

Además aprovecharemos para activar el soporte para claves foráneas, que habíamos olvidado hacer:

```
PRAGMA foreign_keys = ON;
```

La función queda así:

```
bool VerificarTablas(sqlite3 *db) {
    char consulta[36];
    char *tabla[] = {
        "editorial",
        "libro",
        "autor",
        "tema",
        "ejemplar",
        "socio",
        "prestamo",
        "trata_sobre",
        "escrito_por"
    };
    char *create[] = {
        "CREATE TABLE editorial("
          "claveeditorial INTEGER PRIMARY KEY,"
          "editorial TEXT,"
          "direccion TEXT,"
          "telefono TEXT);",
        "CREATE TABLE libro("
          "clavelibro INTEGER PRIMARY KEY,"
          "titulo TEXT,"
          "idioma TEXT,"
          "formato TEXT,"
          "claveeditorial INTEGER "
```

```
          "REFERENCES editorial(claveeditorial) "
          "ON DELETE SET NULL "
          "ON UPDATE CASCADE);",
        "CREATE TABLE autor("
          "claveautor INTEGER PRIMARY KEY,"
          "autor TEXT);",
        "CREATE TABLE tema("
          "clavetema INTEGER PRIMARY KEY,"
          "tema TEXT);",
        "CREATE TABLE ejemplar("
          "clavelibro INTEGER "
          "REFERENCES libro(clavelibro) "
          "ON DELETE CASCADE "
          "ON UPDATE CASCADE,"
          "numeroorden INTEGER NOT NULL,"
          "edicion INTEGER,"
          "ubicacion TEXT,"
          "categoria TEXT,"
          "PRIMARY KEY(clavelibro,numeroorden));",
        "CREATE TABLE socio("
          "clavesocio INTEGER PRIMARY KEY,"
          "socio TEXT,"
          "direccion TEXT,"
          "telefono TEXT,"
          "categoria TEXT);",
        "CREATE TABLE prestamo("
          "clavesocio INTEGER "
          "REFERENCES socio(clavesocio) "
          "ON DELETE SET NULL "
          "ON UPDATE CASCADE,"
          "clavelibro INTEGER "
          "REFERENCES ejemplar(clavelibro) "
          "ON DELETE SET NULL "
          "ON UPDATE CASCADE,"
          "numeroorden INTEGER,"
          "fecha_prestamo DATE NOT NULL,"
          "fecha_devolucion DATE DEFAULT NULL,"
          "notas TEXT);",
        "CREATE TABLE trata_sobre("
          "clavelibro INTEGER NOT NULL "
          "REFERENCES libro(clavelibro) "
          "ON DELETE CASCADE "
          "ON UPDATE CASCADE,"
          "clavetema INTEGER NOT NULL "
          "REFERENCES tema(clavetema) "
          "ON DELETE CASCADE "
          "ON UPDATE CASCADE,"
                 "UNIQUE(clavelibro,clavetema));",
```

```
        "CREATE TABLE escrito_por("
          "clavelibro INTEGER NOT NULL "
          "REFERENCES libro(clavelibro) "
          "ON DELETE CASCADE "
          "ON UPDATE CASCADE,"
          "claveautor INTEGER NOT NULL "
          "REFERENCES autor(claveautor) "
          "ON DELETE CASCADE "
          "ON UPDATE CASCADE,"
                 "UNIQUE(clavelibro,claveautor));"
    };

    // Activar soporte para claves foráneas
    if(SQLITE_OK != sqlite3_exec(db, "PRAGMA foreign_keys =
ON;", 0, 0, 0)) {
        cout << "Imposible activar claves foraneas" << endl;
        return false;
    }

    for(int i = 0; i < nTablas; i++) {
        sprintf(consulta, "SELECT COUNT(*) FROM %s;",
tabla[i]);
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "La tabla " << tabla[i] << " no existe."
<< endl;
            if(SQLITE_OK != sqlite3_exec(db, create[i], 0,
0, 0)) {
               cout << "Error al crear la tabla " <<
tabla[i] << endl;
               return false;
            }
        }
    }
    return true;
}
```

Tratamiento de libros

Por último, añadimos los ficheros "libro.h" y "libro.cpp" al proyecto. En este caso necesitamos algunas funciones más, y el código se complica un poco:

Fichero "libro.h":

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: libro.h
 * fichero de cabecera para manupular datos de libros
 */

#ifndef __LIBRO_H__
#define __LIBRO_H__

#include <sqlite/sqlite3.h>

void NuevoLibro(sqlite3 *);
int ListaLibros(sqlite3 *);
void EditarLibro(sqlite3 *);
void BorrarLibro(sqlite3 *);
void BuscarLibro(sqlite3 *);
void AnexarAutor(sqlite3 *);
void AnexarTema(sqlite3 *);
int ListaAutoresLibro(sqlite3 *, int clavelibro);
void EliminarAutorLibro(sqlite3 *);
int ListaTemasLibro(sqlite3 *, int clavelibro);
void EliminarTemaLibro(sqlite3 *);
#endif
```

Fichero "libro.cpp":

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: libro.cpp
 * fichero de implementación para manipular datos de libros
 */

#include <iostream>
#include <iomanip>
#include "libro.h"
#include "editorial.h"
#include "autor.h"
```

```
#include "tema.h"

using namespace std;

void NuevoLibro(sqlite3 *db) {
    char titulo[64];
    char idioma[64];
    char formato[32];
    int editorial;
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    bool existe, ignorar=false;
    char titulo2[64];
    int clave;
    char resp[2];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira el titulo, idioma,
formato y editorial del libro." << endl;

    cin.ignore();
    cout << "Titulo: ";
    cin.getline(titulo, 64);
    cout << "Idioma: ";
    cin.getline(idioma,64);
    cout << "Formato: ";
    cin.getline(formato,32);
    // Seleccionar editorial:
    editorial = ListaEditoriales(db);

    // Verificar si el nombre existe ya:
    sprintf(consulta, "SELECT clavelibro,titulo FROM libro
WHERE titulo LIKE '%s';", titulo);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    existe=false;
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            existe = true;
            clave = sqlite3_column_int(ppStmt, 0);
            strncpy(titulo2, (const
char*)sqlite3_column_text(ppStmt, 1), 64);
            titulo2[63]=0;
        }
        sqlite3_finalize(ppStmt);
```

```
    }

    if(!existe) {
        sprintf(consulta, "INSERT INTO
libro(titulo,idioma,formato,claveeditorial)
VALUES('%s','%s','%s',%d);",
                titulo, idioma, formato, editorial);
    } else {
        cout << "Ya existe un libro con el titulo " <<
titulo2 << " (s)obrescribir, insert(a)r o (i)gnorar: " <<
endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                sprintf(consulta, "UPDATE libro SET
titulo='%s',idioma='%s',formato='%s',claveeditorial=%d WHERE
clavelibro=%d;",
                        titulo, idioma, formato, editorial,
clave);
                break;
            case 'a':
                sprintf(consulta, "INSERT INTO
libro(titulo,idioma,formato,claveeditorial)
VALUES('%s','%s','%s','%d);",
                        titulo, idioma, formato, editorial);
                break;
            case 'i':
            default:
                ignorar=true;
        }
    }
    if(!ignorar) {
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Libro insertado" << endl;
    }
    cin.ignore();
}

int ListaLibros(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
```

```
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir libro" << endl << endl;
        sprintf(consulta, "SELECT clavelibro,titulo FROM
libro ORDER BY titulo LIMIT 20 OFFSET %d;", desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);
```

```
    return fila;
}

void EditarLibro(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char titulo[64];
    char idioma[64];
    char formato[32];
    int editorial;
    int i;
    int fila;
    bool salir=true;

    fila = ListaLibros(db);

    // Editar:
    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT
titulo,idioma,formato,claveeditorial,editorial FROM libro
NATURAL LEFT JOIN editorial WHERE clavelibro='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Titulo:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Idioma:    " <<
sqlite3_column_text(ppStmt, 1) << endl;
            cout << "Formato:   " <<
sqlite3_column_text(ppStmt, 2) << endl;
            cout << "Editorial: " <<
sqlite3_column_text(ppStmt, 5) << endl;
            cout << "Dejar en blanco los campos que no se
quieren modifiar" << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Titulo: ";
        cin.getline(titulo, 64);
        cout << "Idioma: ";
```

```
        cin.getline(idioma,64);
        cout << "Formato: ";
        cin.getline(formato,32);
        editorial = ListaEditoriales(db);
        if(strlen(titulo)>0) {
            sprintf(consulta, "UPDATE libro SET titulo='%s'
WHERE clavelibro=%d;", titulo, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(idioma)>0) {
            sprintf(consulta, "UPDATE libro SET idioma='%s'
WHERE clavelibro=%d;", idioma, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(formato)>0) {
            sprintf(consulta, "UPDATE libro SET formato='%s'
WHERE clavelibro=%d;", formato, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(editorial) {
            sprintf(consulta, "UPDATE libro SET
claveeditorial=%d WHERE clavelibro=%d;", editorial, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        cout << "Libro modificado" << endl;
        cin.ignore();
    }
}

void BorrarLibro(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
```

```
    char consulta[1024];
    int fila;
    char resp[2];
    int i;
    bool salir=true;

    fila = ListaLibros(db);
    cout << "Borrar: " << fila << endl;

    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT titulo,idioma,formato FROM
editorial WHERE clavelibro='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Titulo:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Idioma:    " <<
sqlite3_column_text(ppStmt, 1) << endl;
            cout << "Formato:   " <<
sqlite3_column_text(ppStmt, 2) << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Borrar este registro? (s/n)" << endl;
        cin >> resp;
        if(resp[0] == 's' || resp[0] == 'S') {
            sprintf(consulta, "DELETE FROM libro WHERE
clavelibro=%d;", fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
    }
    cout << "Libro borrado" << endl;
    cin.ignore();
}

void BuscarLibro(sqlite3 *db) {
```

```
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char filtro[256];
    char titulo[64];
    char idioma[64];
    char formato[32];
    char editorial[32];
    char autor[64];
    char tema[32];
    int i, colautor=5, coltema=5;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Introducir cadenas de busqueda, _ para comodin
de caracter, % para comodin de cadena" << endl;
    cout << "Dejar en blanco para ignorar el campo en la
busqueda" << endl;
    cin.ignore();
    cout << "Titulo: ";
    cin.getline(titulo, 64);
    cout << "Idioma: ";
    cin.getline(idioma,64);
    cout << "Formato: ";
    cin.getline(formato,32);
    cout << "Editorial: ";
    cin.getline(editorial,32);
    cout << "Autor: ";
    cin.getline(autor,64);
    cout << "Tema: ";
    cin.getline(tema,32);

    if(strlen(titulo) == 0) strcpy(titulo, "%");
    if(strlen(idioma) == 0) strcpy(idioma, "%");
    if(strlen(formato) == 0) strcpy(formato, "%");
    if(strlen(editorial) == 0) strcpy(editorial, "%");
    strcpy(consulta, "SELECT
titulo,idioma,formato,editorial");
    if(strlen(autor) > 0) { coltema++; strcat(consulta,
",autor"); }
    if(strlen(tema) > 0) strcat(consulta, ",tema");
    strcat(consulta, " FROM libro NATURAL LEFT JOIN
editorial ");
    if(strlen(autor) > 0) strcat(consulta, "NATURAL JOIN
escrito_por NATURAL JOIN autor ");
    if(strlen(tema)> 0) strcat(consulta, "NATURAL JOIN
trata_sobre NATURAL JOIN tema ");
```

```
    sprintf(filtro, "WHERE titulo LIKE '%s' AND idioma LIKE
'%s' AND formato LIKE '%s' "
            "AND editorial LIKE '%s'",
            titulo, idioma, formato, editorial);
    strcat(consulta, filtro);
    if(strlen(autor) > 0) {
        sprintf(filtro, " AND autor LIKE '%s'", autor);
        strcat(consulta, filtro);
    }
    if(strlen(tema) > 0) {
        sprintf(filtro, " AND tema LIKE '%s'", tema);
        strcat(consulta, filtro);
    }
    strcat(consulta, ";");

    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << "\n" <<
consulta << endl;
    } else {
        i = 0;
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout.setf(ios::left);
            cout.width(64);
            cout << sqlite3_column_text(ppStmt, 0) << endl;
            cout << sqlite3_column_text(ppStmt, 1) << " ";
            cout.width(32);
            cout << sqlite3_column_text(ppStmt, 2) << endl;
            cout.width(32);
            cout << sqlite3_column_text(ppStmt, 4) << endl;
            if(strlen(autor)>0) {
                cout.width(64);
                cout << sqlite3_column_text(ppStmt,
colautor) << endl;
                i++;
            }
            if(strlen(tema)>0) {
                cout.width(32);
                cout << sqlite3_column_text(ppStmt, coltema)
<< endl;
                i++;
            }
            i +=3;
            if(i >= 21) {
                cout << "Pulsa return";
                cin.ignore();
                cin.get();
```

```
            }
        }
        sqlite3_finalize(ppStmt);
    }
    cin.ignore();
    cin.get();
}

void AnexarAutor(sqlite3 *db) {
    char consulta[1024];
    int clavelibro, claveautor;
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Seleccionar un libro y un autor de las listas
siguientes:" << endl;
    cin.ignore();
    clavelibro = ListaLibros(db);
    claveautor = ListaAutores(db);

    if(clavelibro && claveautor) {
        sprintf(consulta, "INSERT INTO
escrito_por(clavelibro,claveautor) VALUES(%d,%d);",
clavelibro, claveautor);
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Autor anexado a libro" << endl;
    }
}

void AnexarTema(sqlite3 *db) {
    char consulta[1024];
    int clavelibro, clavetema;
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Seleccionar un libro y un tema de las listas
siguientes:" << endl;
    cin.ignore();
    clavelibro = ListaLibros(db);
    clavetema = ListaTemas(db);
```

```
    if(clavelibro && clavetema) {
        sprintf(consulta, "INSERT INTO
trata_sobre(clavelibro,clavetema) VALUES(%d,%d);",
clavelibro, clavetema);
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Tema anexado a libro" << endl;
    }
}

int ListaAutoresLibro(sqlite3 *db, int clavelibro) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir autor" << endl << endl;
        sprintf(consulta, "SELECT autor.claveautor,autor
FROM escrito_por NATURAL JOIN autor "
                "WHERE clavelibro=%d ORDER BY autor LIMIT 20
OFFSET %d;", clavelibro, desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
```

```
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
}

void EliminarAutorLibro(sqlite3 *db) {
    char consulta[1024];
    int clavelibro, claveautor;
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Seleccionar un libro:" << endl;
    cin.ignore();
    clavelibro = ListaLibros(db);
    if(clavelibro) {
        claveautor = ListaAutoresLibro(db, clavelibro);
        if(claveautor) {
            sprintf(consulta, "DELETE FROM escrito_por WHERE
clavelibro=%d AND claveautor=%d;", clavelibro, claveautor);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
            else cout << "Autor eliminado de libro" << endl;
        }
    }
```

```
}

int ListaTemasLibro(sqlite3 *db, int clavelibro) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir tema" << endl << endl;
        sprintf(consulta, "SELECT tema.clavetema,tema FROM
trata_sobre NATURAL JOIN tema "
                "WHERE clavelibro=%d ORDER BY tema LIMIT 20
OFFSET %d;", clavelibro, desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
```

```
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
}

void EliminarTemaLibro(sqlite3 *db) {
    char consulta[1024];
    int clavelibro, clavetema;
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Seleccionar un libro:" << endl;
    cin.ignore();
    clavelibro = ListaLibros(db);
    if(clavelibro) {
        clavetema = ListaTemasLibro(db, clavelibro);
        if(clavetema) {
            sprintf(consulta, "DELETE FROM trata_sobre WHERE
clavelibro=%d AND clavetema=%d;", clavelibro, clavetema);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
            else cout << "Tema eliminado de libro" << endl;
        }
    }
}
```

Procesar tabla de ejemplares

A veces encontramos errores de diseño en la fase de codificación. Este es uno de esos casos, al intentar eliminar

cualquier fila de la tabla de ejemplares obtenemos un error, o más bien, una excepción:

```
sqlite> DELETE FROM ejemplar;
Error: foreign key mismatch
sqlite>
```

Esto se debe a un error de diseño de las tablas de la base de datos. Concretamente, en la tabla de préstamos usamos un campo *clavelibro* que es una clave foránea de la tabla *ejemplar*:

```
 ... clavelibro INTEGER REFERENCES ejemplar(clavelibro) ON
DELETE ...
```

Sería más lógico usar como clave foránea una columna de la tabla *ejemplar* que no fuese a su vez una clave foránea de la tabla *libro*. Esto nos obliga a crear una nueva columna en la tabla *ejemplar*, *claveejemplar* que será además su clave primaria, y que usaremos para referenciar el ejemplar desde la tabla de *prestamo*.

Al mismo tiempo, crearemos una restricción de unicidad para la combinación *clavelibro*/*numeroorden*, para que no puedan existir dos ejemplares iguales.

Las tablas *ejemplar* y *prestamo* quedan así:

```
CREATE TABLE ejemplar(claveejemplar INTEGER PRIMARY KEY,
    clavelibro INTEGER NOT NULL
    REFERENCES libro(clavelibro)
    ON DELETE CASCADE ON UPDATE CASCADE,
    numeroorden INTEGER NOT NULL,
    edicion INTEGER,
    ubicacion TEXT,
    UNIQUE(numeroorden,clavelibro));
CREATE TABLE prestamo(clavesocio INTEGER REFERENCES
socio(clavesocio)
    ON DELETE SET NULL ON UPDATE CASCADE,
    claveejemplar INTEGER REFERENCES ejemplar(claveejemplar)
```

```
    ON DELETE SET NULL ON UPDATE CASCADE,
    fecha_prestamo DATE NOT NULL,
    fecha_devolucion DATE DEFAULT NULL,
    notas TEXT);
```

En cuanto a las modificaciones, hacemos como en los casos anteriores. Primero añadimos los identificadores de menú necesarios en "menus.h":

```
#define NUEVOEJEMPLAR       22
#define EDITAREJEMPLAR      23
#define BORRAREJEMPLAR      24
#define CONSULTAEJEMPLAR    25
```

También añadimos las nuevas entradas en la tabla de menús, en menus.cpp:

```
    {9, "-", "---MENU EJEMPLARES---",0,TITULO},
    {9, "1", "Nuevo", 0, NUEVOEJEMPLAR},
    {9, "2", "Editar", 0, EDITAREJEMPLAR},
    {9, "3", "Borrar", 0, BORRAREJEMPLAR},
    {9, "4", "Consultar", 0, CONSULTAEJEMPLAR},
    {9, "0", "Salir <", 3, ABRIRMENU}
```

En main.cpp añadimos el include del nuevo fichero de cabecera (ejemplar.h), y los tratamientos de las nuevas opciones de menú dentro del *switch*:

```
#include "ejemplar.h"
...
            case NUEVOEJEMPLAR:
               NuevoEjemplar(db);
               break;
            case EDITAREJEMPLAR:
               EditarEjemplar(db);
               break;
            case BORRAREJEMPLAR:
```

```
                BorrarEjemplar(db);
                break;
            case CONSULTAEJEMPLAR:
                BuscarEjemplar(db);
                break;
```

Añadimos dos nuevos ficheros al proyecto: ejemplar.h, con las declaraciones de prototipos, y ejemplar.cpp con la implementación de las funciones:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: ejemplar.h
 * fichero de cabecera para manupular datos de ejemplares de
libros
 */

#ifndef __EJEMPLAR_H__
#define __EJEMPLAR_H__

#include <sqlite/sqlite3.h>

void NuevoEjemplar(sqlite3 *);
int ListaEjemplares(sqlite3 *);
void EditarEjemplar(sqlite3 *);
void BorrarEjemplar(sqlite3 *);
void BuscarEjemplar(sqlite3 *);
#endif
```

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: ejemplar.cpp
 * fichero de implementación para manipular datos de
ejemplares de libros
 */
```

```
#include <iostream>
#include <iomanip>
#include "ejemplar.h"
#include "libro.h"

using namespace std;

void NuevoEjemplar(sqlite3 *db) {
    int clavelibro;
    int numeroorden;
    int edicion;
    char ubicacion[32];
    char categoria[2]; // "A".."F"
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    bool existe, ignorar=false;
    char titulo[64];
    char numero[15];
    char resp[2];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira el libro, numero de
orden, edicion, ubicacion y categoría del ejemplar." <<
endl;

    cin.ignore();
    cout << "Libro: ";
    clavelibro = ListaLibros(db);
    cin.ignore();
    cout << "Numero de orden: ";
    cin.getline(numero,15);
    numeroorden = atoi(numero);
    cout << "Edicion: ";
    cin.getline(numero,15);
    edicion = atoi(numero);
    cout << "Ubicacion: ";
    cin.getline(ubicacion,32);
    cout << "Categoria: ";
    cin.getline(categoria,2);

    // Verificar si el nombre existe ya:
    sprintf(consulta, "SELECT titulo FROM ejemplar NATURAL
JOIN libro "
            "WHERE clavelibro=%d AND numeroorden=%d;",
clavelibro, numeroorden);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
```

```
NULL);
    existe=false;
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            existe = true;
            strncpy(titulo, (const
char*)sqlite3_column_text(ppStmt, 0), 64);
            titulo[63]=0;
        }
        sqlite3_finalize(ppStmt);
    }

    if(!existe) {
        sprintf(consulta, "INSERT INTO
ejemplar(clavelibro,numeroorden,edicion,ubicacion,categoria)
VALUES(%d,%d,%d,'%s','%c');",
                clavelibro, numeroorden, edicion, ubicacion,
categoria[0]);
    } else {
        cout << "Ya existe un ejemplar del libro [" <<
titulo << "] con el numero de orden [" << numeroorden;
        cout << "]\n(s)obrescribir, insert(a)r o (i)gnorar:
" << endl;
        cin>> resp;
        switch(resp[0]) {
            case 's':
                sprintf(consulta, "UPDATE ejemplar SET
edicion=%d,ubicacion='%s',categoria='%c' WHERE clavelibro=%d
AND numeroorden=%d;",
                        edicion, ubicacion, categoria[0],
clavelibro, numeroorden);
                break;
            case 'a':
                sprintf(consulta, "INSERT INTO
ejemplar(clavelibro,numeroorden,edicion,ubicacion,categoria)
VALUES(%d,%d,%d,'%s','%c');",
                        clavelibro, numeroorden, edicion,
ubicacion,categoria[0]);
                break;
            case 'i':
            default:
                ignorar=true;
        }
    }
    if(!ignorar) {
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
```

```
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Ejemplar insertado" << endl;
    }
    cin.ignore();
}

int ListaEjemplares(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir ejemplar" << endl << endl;
        sprintf(consulta, "SELECT
claveejemplar,titulo,numeroorden FROM ejemplar NATURAL JOIN
libro "
                "ORDER BY titulo,numeroorden LIMIT 20 OFFSET
%d;", desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << " ["
<<
                    sqlite3_column_int(ppStmt, 2) << "]" <<
endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
```

```
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
}

void EditarEjemplar(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int numeroorden;
    int edicion;
    char ubicacion[32];
    char categoria[2]; // "A".."F"
    char numero1[16];
    char numero2[16];
    int i;
    int fila;
    bool salir=true;

    fila = ListaEjemplares(db);

    // Editar:
    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT
titulo,numeroorden,edicion,ubicacion,categoria FROM ejemplar
NATURAL LEFT JOIN libro WHERE claveejemplar='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
```

```
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Titulo:          " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Numero de orden: " <<
sqlite3_column_int(ppStmt, 1) << endl;
            cout << "Edicion:         " <<
sqlite3_column_int(ppStmt, 2) << endl;
            cout << "Ubicacion:       " <<
sqlite3_column_text(ppStmt, 3) << endl;
            cout << "Categoria:       " <<
sqlite3_column_text(ppStmt, 3) << endl;
            cout << "Dejar en blanco los campos que no se
quieren modifiar" << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Numero de orden: ";
        cin.getline(numero1,15);
        numeroorden = atoi(numero1);
        cout << "Edicion: ";
        cin.getline(numero2,15);
        edicion = atoi(numero2);
        cout << "Ubicacion: ";
        cin.getline(ubicacion,32);
        cout << "Categoria: ";
        cin.getline(categoria,2);
        if(strlen(numero1)>0) {
            sprintf(consulta, "UPDATE ejemplar SET
numeroorden=%d WHERE claveejemplar=%d;", numeroorden, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(numero2)>0) {
            sprintf(consulta, "UPDATE ejemplar SET
edicion=%d WHERE claveejemplar=%d;", edicion, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
```

```
            }
        }
        if(strlen(ubicacion)>0) {
            sprintf(consulta, "UPDATE ejemplar SET
ubicacion='%s' WHERE claveejemplar=%d;", ubicacion, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(categoria)>0) {
            sprintf(consulta, "UPDATE ejemplar SET
categoria='%c' WHERE clavelibro=%d;", categoria[0], fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        cout << "Ejemplar modificado" << endl;
        cin.ignore();
    }
}

void BorrarEjemplar(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int fila;
    char resp[2];
    int i;
    bool salir=true;

    fila = ListaEjemplares(db);
    cout << "Borrar: " << fila << endl;

    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT
titulo,numeroorden,edicion,ubicacion,categoria FROM ejemplar
NATURAL LEFT JOIN libro WHERE claveejemplar='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
```

```
            cout << "Titulo:          " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Numero de orden: " <<
sqlite3_column_int(ppStmt, 1) << endl;
            cout << "Edicion:         " <<
sqlite3_column_int(ppStmt, 2) << endl;
            cout << "Ubicacion:       " <<
sqlite3_column_text(ppStmt, 3) << endl;
            cout << "Categoria:       " <<
sqlite3_column_text(ppStmt, 4) << endl;
            cout << "Dejar en blanco los campos que no se
quieren modifiar" << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Borrar este registro? (s/n)" << endl;
        cin >> resp;
        if(resp[0] == 's' || resp[0] == 'S') {
            sprintf(consulta, "DELETE FROM ejemplar WHERE
claveejemplar=%d;", fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
    }
    cout << "Ejemplar borrado" << endl;
    cin.ignore();
}

void BuscarEjemplar(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char filtro[256];
    char titulo[64];
    char numero1[16];
    char numero2[16];
    int edicion;
    int numeroorden;
    char ubicacion[32];
    char categoria[2];
    int i;
```

```
    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de editoriales por nombre, direccion o
telefono:
    cout << "Introducir cadenas de busqueda, _ para comodin
de caracter, % para comodin de cadena" << endl;
    cout << "Dejar en blanco para ignorar el campo en la
busqueda" << endl;
    cin.ignore();
    cout << "Titulo: ";
    cin.getline(titulo, 64);
    cout << "Numero de orden: ";
    cin.getline(numero1,15);
    numeroorden = atoi(numero1);
    cout << "Edicion: ";
    cin.getline(numero2,15);
    edicion = atoi(numero2);
    cout << "Ubicacion: ";
    cin.getline(ubicacion,2);
    cout << "Categoria: ";
    cin.getline(categoria,2);

    if(strlen(titulo) == 0) strcpy(titulo, "%");
    if(strlen(ubicacion) == 0) strcpy(ubicacion, "%");
    if(strlen(categoria) == 0) strcpy(categoria, "%");
    sprintf(consulta, "SELECT
titulo,numeroorden,edicion,ubicacion,categoria FROM ejemplar
NATURAL JOIN libro "
            "WHERE titulo LIKE '%s' AND ubicacion LIKE '%s'
AND categoria LIKE '%s'", titulo, ubicacion, categoria);
    if(strlen(numero1) > 0) {
        sprintf(filtro, " AND numeroorden=%d", numeroorden);
        strcat(consulta, filtro);
    }
    if(strlen(numero2) > 0) {
        sprintf(filtro, " AND edicion=%d", edicion);
        strcat(consulta, filtro);
    }
    strcat(consulta, ";");

    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << "\n" <<
consulta << endl;
    } else {
        i = 0;
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout.setf(ios::left);
```

```
            cout.width(64);
            cout << sqlite3_column_text(ppStmt, 0);
            cout.width(13);
            cout << "[" << sqlite3_column_text(ppStmt, 1) <<
"]" << endl;
            cout << "Ed: " << sqlite3_column_text(ppStmt,
2);
            cout.width(32);
            cout << " Ubic:" << sqlite3_column_text(ppStmt,
3);
            cout.width(2);
            cout << " Cat:" << sqlite3_column_text(ppStmt,
4) << endl;
            i +=2;
            if(i >= 21) {
                cout << "Pulsa return";
                cin.ignore();
                cin.get();
            }
        }
        sqlite3_finalize(ppStmt);
    }
    cin.ignore();
    cin.get();
}
```

Procesar tabla de socios

Una vez más, añadiremos los identificadores para las nuevas opciones de menú en "menus.h":

```
#define NUEVOSOCIO          26
#define EDITARSOCIO         27
#define BORRARSOCIO         28
#define CONSULTASOCIO       29
```

También añadiremos las nuevas opciones de menú para los socios en "menus.cpp", y borraremos la tabla de menú usando la sentencia "DROP TABLE menu;" desde la consola de SQLite:

```
    {4, "-", "---MENU SOCIOS---",0,TITULO},
    {4, "1", "Nuevo", 0, NUEVOSOCIO},
    {4, "2", "Editar", 0, EDITARSOCIO},
    {4, "3", "Borrar", 0, BORRARSOCIO},
    {4, "4", "Consultar", 0, CONSULTASOCIO},
    {4, "0", "Salir <", 1, ABRIRMENU},
```

Volvemos a modificar el fichero "main.cpp", para añadir el fichero incluído "socio.h" y el proceso de las nuevas opciones de menú:

```
#include "socio.h"
...
            case NUEVOSOCIO:
               NuevoSocio(db);
               break;
            case EDITARSOCIO:
               EditarSocio(db);
               break;
            case BORRARSOCIO:
               BorrarSocio(db);
               break;
            case CONSULTASOCIO:
               BuscarSocio(db);
               break;
```

Finalmente, añadiremos dos ficheros más al proyecto, uno con los prototipos de funciones y otro con la implementación para el tratamiento de la tabla de socios:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: socio.h
 * fichero de cabecera para manupular datos de socios
 */

#ifndef __SOCIO_H__
```

```
#define __SOCIO_H__

#include <sqlite/sqlite3.h>

void NuevoSocio(sqlite3 *);
int ListaSocios(sqlite3 *);
void EditarSocio(sqlite3 *);
void BorrarSocio(sqlite3 *);
void BuscarSocio(sqlite3 *);
#endif
```

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: socio.cpp
 * fichero de implementación para manipular datos de socios
 */

#include <iostream>
#include <iomanip>
#include "socio.h"

using namespace std;

void NuevoSocio(sqlite3 *db) {
    char nombre[64];
    char direccion[128];
    char telefono[32];
    char categoria[2];
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    bool existe, ignorar=false;
    char nombre2[64];
    int clave;
    char resp[2];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira el nombre, direccion,
telefono y cateforia del socio." << endl;

    cin.ignore();
    cout << "Nombre: ";
```

```
    cin.getline(nombre, 64);
    cout << "Direccion: ";
    cin.getline(direccion, 128);
    cout << "Telefono: ";
    cin.getline(telefono, 32);
    cout << "Categoria (A-F): ";
    cin.getline(categoria, 2);

    // Verificar si el nombre existe ya:
    sprintf(consulta, "SELECT clavesocio,socio FROM socio
WHERE socio LIKE '%s';", nombre);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    existe=false;
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            existe = true;
            clave = sqlite3_column_int(ppStmt, 0);
            strncpy(nombre2, (const
char*)sqlite3_column_text(ppStmt, 1), 64);
            nombre2[63]=0;
        }
        sqlite3_finalize(ppStmt);
    }

    if(!existe) {
        sprintf(consulta, "INSERT INTO
socio(socio,direccion,telefono,categoria)
VALUES('%s','%s','%s','%c');",
                nombre, direccion, telefono, categoria[0]);
    } else {
        cout << "Ya existe un socio con el nombre " <<
nombre2 << " (s)obrescribir, insert(a)r o (i)gnorar: " <<
endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                sprintf(consulta, "UPDATE socio SET
autor='%s',direccion='%s',telefono='%s',categoria='%c' WHERE
claveautor=%d;",
                        nombre, direccion, telefono,
categoria[0], clave);
                break;
            case 'a':
                sprintf(consulta, "INSERT INTO
socio(socio,direccion,telefono,categoria)
```

```
VALUES('%s','%s','%s','%c');",
                        nombre, direccion, telefono,
categoria[0]);
                break;
            case 'i':
            default:
                ignorar=true;
        }
    }
    if(!ignorar) {
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Socio insertado" << endl;
    }
    cin.ignore();
}

int ListaSocios(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir socio" << endl << endl;
        sprintf(consulta, "SELECT clavesocio,socio FROM
socio ORDER BY socio LIMIT 20 OFFSET %d;", desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << endl;
                i++;
            }
```

```
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
}

void EditarSocio(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    char direccion[128];
    char telefono[32];
    char categoria[2];
    int i;
    int fila;
    bool salir=true;

    fila = ListaSocios(db);

    // Editar:
    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT
socio,direccion,telefono,categoria FROM socio WHERE
clavesocio='%d';", fila);
```

```
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Direccion: " <<
sqlite3_column_text(ppStmt, 1) << endl;
            cout << "Telefono:  " <<
sqlite3_column_text(ppStmt, 2) << endl;
            cout << "Categoria: " <<
sqlite3_column_text(ppStmt, 3) << endl;
            cout << "Dejar en blanco los campos que no se
quieren modifiar" << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Nombre: ";
        cin.getline(nombre, 64);
        cout << "Direccion: ";
        cin.getline(direccion, 128);
        cout << "Telefono: ";
        cin.getline(telefono, 32);
        cout << "Categoria: ";
        cin.getline(categoria, 2);
        if(strlen(nombre)>0) {
            sprintf(consulta, "UPDATE socio SET socio='%s'
WHERE clavesocio=%d;", nombre, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(direccion)>0) {
            sprintf(consulta, "UPDATE socio SET
direccion='%s' WHERE clavesocio=%d;", direccion, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
```

```
        }
        if(strlen(telefono)>0) {
            sprintf(consulta, "UPDATE socio SET
telefono='%s' WHERE clavesocio=%d;", telefono, fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        if(strlen(categoria)>0) {
            sprintf(consulta, "UPDATE socio SET
categoria='%c' WHERE clavesocio=%d;", categoria[0], fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
        cout << "Socio modificado" << endl;
        cin.ignore();
    }
}

void BorrarSocio(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int fila;
    char resp[2];
    int i;
    bool salir=true;

    fila = ListaSocios(db);
    cout << "Borrar: " << fila << endl;

    for(i = 0; i < 22; i++) cout << endl;
    sprintf(consulta, "SELECT
socio,direccion,telefono,categoria FROM socio WHERE
clavesocio='%d';", fila);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout << "Nombre:    " <<
```

```
sqlite3_column_text(ppStmt, 0) << endl;
            cout << "Direccion: " <<
sqlite3_column_text(ppStmt, 1) << endl;
            cout << "Telefono:  " <<
sqlite3_column_text(ppStmt, 2) << endl;
            cout << "Categoria: " <<
sqlite3_column_text(ppStmt, 3) << endl;
            salir=false;
        }
        sqlite3_finalize(ppStmt);
    }
    if(!salir){
        cin.ignore();
        cout << "Borrar este registro? (s/n)" << endl;
        cin >> resp;
        if(resp[0] == 's' || resp[0] == 'S') {
            sprintf(consulta, "DELETE FROM socio WHERE
clavesocio=%d;", fila);
            if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0,
0)) {
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            }
        }
    }
    cout << "Socio borrado" << endl;
    cin.ignore();
}

void BuscarSocio(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char nombre[64];
    char direccion[128];
    char telefono[32];
    char categoria[2];
    int i;

    for(i = 0; i < 22; i++) cout << endl;
    // Búsqueda de Socios por nombre, direccion, telefono o
categoria:
    cout << "Introducir cadenas de busqueda, _ para comodin
de caracter, % para comodin de cadena" << endl;
    cout << "Dejar en blanco para ignorar el campo en la
busqueda" << endl;
    cin.ignore();
    cout << "Nombre: ";
```

```
    cin.getline(nombre, 64);
    cout << "Direccion: ";
    cin.getline(direccion, 128);
    cout << "Telefono: ";
    cin.getline(telefono, 32);
    cout << "Categoria: ";
    cin.getline(categoria, 2);

    if(strlen(nombre) == 0) strcpy(nombre, "%");
    if(strlen(direccion) == 0) strcpy(direccion, "%");
    if(strlen(telefono) == 0) strcpy(telefono, "%");
    if(strlen(categoria) == 0) strcpy(categoria, "%");
    sprintf(consulta, "SELECT
socio,direccion,telefono,categoria FROM socio "
            "WHERE socio LIKE '%s' AND direccion LIKE '%s'
AND telefono LIKE '%s' AND categoria LIKE '%c';",
            nombre, direccion, telefono, categoria[0]);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    } else {
        i = 0;
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            cout.setf(ios::left);
            cout.width(64);
            cout << sqlite3_column_text(ppStmt, 0) << endl;
            cout.width(128);
            cout << sqlite3_column_text(ppStmt, 1) << endl;
            cout.width(32);
            cout << sqlite3_column_text(ppStmt, 2) << " ";
            cout.width(32);
            cout << "Cat: " << sqlite3_column_text(ppStmt,
3) << endl;
            i += 3;
            if((i >= 22)) {
                cout << "Pulsa return";
                cin.ignore();
                cin.get();
            }
        }
        sqlite3_finalize(ppStmt);
    }
    cin.ignore();
    cin.get();
}
```

Procesar tabla de préstamos

Bueno, esta es la última tabla, pero no se ajusta al mismo patrón que las anteriores.

Con los préstamos las opereaciones son algo diferentes, al menos si queremos que el programa resulte operativo según el funcionamiento que vamos a suponer para la biblioteca.

Los préstamos siempre empiezan con un socio que pide un libro. Así, para crear una fila en la tabla de préstamos tenemos que obtener una clave de socio, y una clave de ejemplar, generalmente a través de una clave de libro. En las altas de préstamos dejaremos la fecha de devolución a NULL y la de entrega será, por defecto, la fecha actual, aunque pueda ser modificada por el usuario.

La función *ListaEjemplares* que teníamos definida en "ejemplar.cpp" no nos sirve para esta tarea, ya que necesitamos que nos deje elegir sólo entre los ejemplares que no han sido prestados, que son los disponibles en este caso. Por lo tanto, crearemos una función para seleccionar el ejemplar adecuado.

Esta tarea no es tan simple como puede parecer en principio, ya que no existe una consulta que nos de estos datos de forma directa.

Para conseguirlo primero crearemos una tabla temporal en la que insertaremos los ejemplares existentes del libro seleccionado. Una vez hecho esto, eliminaremos de esa tabla los ejemplares que estén en préstamo (aquellos cuya fecha de devolución sea nula). Podemos eliminar también los ejemplares que no puedan ser prestados al socio, debido a la categoría. Por ejemplo, un ejemplar de categoría C sólo puede ser prestado a socios de categoria B y C. (Recordemos que no puede haber socios de categoría A.

Además, una vez identificado el socio, podemos mostrar una lista de los ejemplares que tiene en préstamo actualmente, para ver si, por ejemplo, ya tiene prestados el número límite (si es que existe un límite).

Los registros de préstamos, en principio, no deberían poderse eliminar ni editar. Pero si se produce algún error en el préstamo

podemos añadir una opción de anular, y podemos dejar editar algunos campos en los préstamos cerrados, por ejemplo, la fecha de préstamo.

Para editar y anular préstamos tendremos que elegirlos desde una lista de préstamos abiertos de un socio. Pero la tabla de préstamos no tiene una clave primaria arbitraria, de manera que necesitamos un valor único, a ser posible numérico, que identifique a cada préstamo.

Afortunadamente, cada ejemplar sólo puede estar prestado una vez en cada momento, por lo que podremos usar la clave de ejemplar para seleccionar el préstamo que queramos editar o anular.

Las devoluciones las haremos a partir de un socio, y una vez conocido éste, mostraremos una lista de los préstamos actuales. Se nos pedirá una fecha de devolución y podremos agregar alguna nota. De hecho, esta función será muy parecida a las de anular y modificar fecha.

Otras posibles opciones son la consulta de historiales de préstamos, tanto de un socio como de un libro.

También nos puede interesar un listado de préstamos que lleven abiertos más de *n* días.

Y en general, cualquier consulta que se nos ocurra que puede ser útil. De momento nos quedaremos con las anteriores.

En cualquier caso, el procedimiento para agregar las nuevas opciones es similar a los anteriores. Empezamos por crear los nuevos identificadores, en "menus.h":

```
#define NUEVOPRESTAMO       30
#define EDITARPRESTAMO      31
#define ANULARPRESTAMO      32
#define DEVOLUCION          33
#define EDITARDEVOLUCION    34
#define HISTORIALSOCIO      35
#define HISTORIALLIBRO      36
#define CONSULTAPRESTAMOS   37
```

También añadiremos las nuevas opciones de menú en "menus.cpp", y borraremos la tabla de "menu" para que se vuelva a generar completa:

```
    {5, "-", "---MENU PRESTAMOS---",0,TITULO},
    {5, "1", "Nuevo prestamo", 0, NUEVOPRESTAMO},
    {5, "2", "Editar prestamo", 0, EDITARPRESTAMO},
    {5, "3", "Anular prestamo", 0, ANULARPRESTAMO},
    {5, "4", "Devolucion", 0, DEVOLUCION},
    {5, "5", "Historial de libro", 0, HISTORIALLIBRO},
    {5, "6", "Historial de socio", 0, HISTORIALSOCIO},
    {5, "7", "Consultar prestamos", 0, CONSULTAPRESTAMOS},
    {5, "0", "Salir <", 1, ABRIRMENU},
```

A continuación, modificaremos el fichero "main.cpp" para que se incluya el nuevo fichero de cabecera "prestamo.h", y para que se procesen las nuevas opciones:

```
#include "prestamo.h"
...
            case NUEVOPRESTAMO:
               NuevoPrestamo(db);
               break;
            case EDITARPRESTAMO:
               CambiarFechaPrestamo(db);
               break;
            case ANULARPRESTAMO:
               AnularPrestamo(db);
               break;
            case DEVOLUCION:
               DevolucionPrestamo(db);
               break;
            case HISTORIALLIBRO:
               HistorialLibro(db);
               break;
            case HISTORIALSOCIO:
               HistorialSocio(db);
               break;
            case CONSULTAPRESTAMOS:
               ConsultaPrestamos(db);
               break;
```

Por último, añadimos dos nuevos ficheros al proyecto: 'prestamo.h' y 'prestamo.cpp', con su consiguiente contenido:

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: prestamo.h
 * fichero de cabecera para manupular datos de prestamos
 */

#ifndef __PRESTAMO_H__
#define __PRESTAMO_H__

#include <sqlite/sqlite3.h>

void NuevoPrestamo(sqlite3 *);
void MostrarEjemplaresPrestados(sqlite3 *, int);
char LeerCategoriaSocio(sqlite3 *, int);
int ListaEjemplaresBiblioteca(sqlite3 *, int, int);
void CambiarFechaPrestamo(sqlite3 *);
void AnularPrestamo(sqlite3 *);
int ListaEjemplaresLibro(sqlite3 *db, int clavelibro);
int ListaEjemplaresPrestados(sqlite3 *, int);
void DevolucionPrestamo(sqlite3 *);
void HistorialLibro(sqlite3 *);
void HistorialSocio(sqlite3 *);
void ConsultaPrestamos(sqlite3 *);
#endif
```

```
/*
 * Aplicación de ejemplo de uso de SQLite en C++
 * EjemploSQLite
 * Salvador Pozo, Con Clase (www.conclase.net)
 * Abril de 2012
 * Fichero: prestamo.cpp
 * fichero de implementación para manipular datos de
prestamos
 */
```

```
#include <iostream>
#include <iomanip>
#include "prestamo.h"
#include "socio.h"
#include "libro.h"
#include "ejemplar.h"

using namespace std;

void NuevoPrestamo(sqlite3 *db) {
    int clavesocio;
    int clavelibro;
    int claveejemplar;
    char fecha[12];
    char consulta[1024];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira un socio, un libro y
un ejemplar para dar de alta un prestamo." << endl;

    cin.ignore();
    cout << "Socio: ";
    clavesocio = ListaSocios(db);
    if(!clavesocio) return;
    MostrarEjemplaresPrestados(db, clavesocio);
    cout << "Libro:";
    clavelibro = ListaLibros(db);
    if(!clavelibro) return;
    claveejemplar = ListaEjemplaresBiblioteca(db,
clavesocio, clavelibro);
    if(!claveejemplar) {
        cout << "No hay ejemplares disponibles para
prestamo" << endl;
        cin.ignore();
        cin.get();
        return;
    }
    cin.ignore();
    cout << "Fecha de entrega (AAAA/MM/DD) [dejar en blanco
para fecha actual]: ";
    cin.getline(fecha, 12);

    if(clavesocio && claveejemplar) {
        if(strlen(fecha) > 0)
            sprintf(consulta, "INSERT INTO
prestamo(clavesocio,claveejemplar,fecha_prestamo,fecha_devol
ucion,notas) VALUES(%d,%d,'%s',NULL,'');",
                clavesocio, claveejemplar, fecha);
```

```
        else
            sprintf(consulta, "INSERT INTO
prestamo(clavesocio,claveejemplar,fecha_prestamo,fecha_devol
ucion,notas) VALUES(%d,%d,date('now'),NULL,'');",
                clavesocio, claveejemplar);
        if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0))
{
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        }
        else cout << "Prestamo creado" << endl;
    }
    cin.ignore();
}

void MostrarEjemplaresPrestados(sqlite3 *db, int clavesocio)
{
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];

    if(clavesocio) {
        sprintf(consulta, "SELECT
fecha_prestamo,titulo,formato FROM prestamo NATURAL JOIN
ejemplar NATURAL JOIN libro "
                "WHERE clavesocio=%d AND fecha_devolucion IS
NULL;", clavesocio);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            cout << "Este socio tiene prestados actualmente
los siguientes ejemplares:" << endl;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_text(ppStmt, 0) << "
" <<
                    sqlite3_column_text(ppStmt, 1) << " ("
<<
                    sqlite3_column_text(ppStmt, 2) << ")" <<
endl;
            }
            cout << "------------------------" << endl;
            sqlite3_finalize(ppStmt);
        }
    }
    cin.ignore();
    cin.get();
}
```

```
char LeerCategoriaSocio(sqlite3 *db, int clavesocio) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[128];
    char categoriasocio[2];

    sprintf(consulta, "SELECT categoria FROM socio WHERE
clavesocio=%d;", clavesocio);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        return 0;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            strncpy(categoriasocio,
(char*)sqlite3_column_text(ppStmt, 0), 1);
            categoriasocio[2] = 0;
        }
        sqlite3_finalize(ppStmt);
    }
    return categoriasocio[0];
}

int ListaEjemplaresBiblioteca(sqlite3 *db, int clavesocio,
int clavelibro) {
    sqlite3_stmt *ppStmt;
    sqlite3_stmt *ppStmt2;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Crear tabla temporal con los ejemplares del libro
indicado:
    sprintf(consulta, "CREATE TEMPORARY TABLE ejemplar2 AS
SELECT * FROM ejemplar WHERE clavelibro=%d;", clavelibro);
    if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0)) {
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        return 0;
    }
    // Eliminar ejemplares de categorias superiores a
categoriasocio:
```

```
    sprintf(consulta, "DELETE FROM ejemplar2 WHERE
categoria<'%c';", LeerCategoriaSocio(db, clavesocio));
    if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0)) {
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        return 0;
    }

    // Eliminar ejemplares prestados:
    sprintf(consulta, "SELECT claveejemplar FROM prestamo
NATURAL JOIN ejemplar NATURAL JOIN libro "
            "WHERE clavelibro=%d AND fecha_devolucion IS
NULL AND claveejemplar IS NOT NULL;", clavelibro);
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc==SQLITE_OK ) {
        rc = sqlite3_prepare_v2(db, "DELETE FROM ejemplar2
WHERE claveejemplar=@clave;", -1, &ppStmt2, NULL);
    }
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        return 0;
    } else {
        while(SQLITE_ROW == sqlite3_step(ppStmt)) {
            sqlite3_bind_int(ppStmt2,
sqlite3_bind_parameter_index(ppStmt2, "@clave"),
sqlite3_column_int(ppStmt, 0));
            sqlite3_step(ppStmt2);
            sqlite3_reset(ppStmt2);
        }
        sqlite3_finalize(ppStmt);
    }

    // Si la lista de ejemplares disponibles está vacía,
retornar 0:
    sprintf(consulta, "SELECT COUNT(*) FROM ejemplar2;");
    rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
    if( rc!=SQLITE_OK ){
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        return 0;
    } else {
        if(SQLITE_ROW == sqlite3_step(ppStmt)) {
            if(sqlite3_column_int(ppStmt, 0) == 0) {
                salir = true;
            }
        }
        sqlite3_finalize(ppStmt);
    }
```

```
    if(salir) {
        sqlite3_exec(db, "DROP TABLE ejemplar2;", 0, 0, 0);
        return 0;
    }

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir ejemplar" << endl << endl;
        sprintf(consulta, "SELECT
claveejemplar,titulo,numeroorden FROM ejemplar2 NATURAL JOIN
libro "
                "ORDER BY titulo,numeroorden LIMIT 20 OFFSET
%d;", desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << " ["
<<
                    sqlite3_column_int(ppStmt, 2) << "]" <<
endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
```

```
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);
    // Borrar tabla temporal:
    sqlite3_exec(db, "DROP TABLE ejemplar2;", 0, 0, 0);

    return fila;
}

void CambiarFechaPrestamo(sqlite3 *db) {
    int clavesocio;
    int claveejemplar;
    char fecha[12];
    char consulta[1024];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira un socio, y se podrá
editar la fecha de entrega de uno de sus prestamos
abiertos." << endl;

    cin.ignore();
    cout << "Socio: ";
    clavesocio = ListaSocios(db);
    if(!clavesocio) return;
    claveejemplar = ListaEjemplaresPrestados(db,
clavesocio);
    if(!claveejemplar) {
        cout << "No hay prestamos abiertos para este socio"
<< endl;
        cin.ignore();
        cin.get();
        return;
    }
    cin.ignore();
    cout << "Fecha de entrega (AAAA/MM/DD) [dejar en blanco
para fecha actual]: ";
    cin.getline(fecha, 12);
    sprintf(consulta, "UPDATE prestamo SET
fecha_prestamo='%s' WHERE claveejemplar=%d AND
fecha_devolucion IS NULL;", fecha, claveejemplar);
    if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0)) {
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    }
}
```

```
void AnularPrestamo(sqlite3 *db) {
    int clavesocio;
    int claveejemplar;
    char consulta[1024];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira un socio, y se podrá
editar la fecha de entrega de uno de sus prestamos
abiertos." << endl;

    cin.ignore();
    cout << "Socio: ";
    clavesocio = ListaSocios(db);
    if(!clavesocio) return;
    claveejemplar = ListaEjemplaresPrestados(db,
clavesocio);
    if(!claveejemplar) {
        cout << "No hay prestamos abiertos para este socio"
<< endl;
        cin.ignore();
        cin.get();
        return;
    }
    cin.ignore();
    // Borrar fila:
    sprintf(consulta, "DELETE FROM prestamo WHERE
claveejemplar=%d AND fecha_devolucion IS NULL;",
claveejemplar);
    if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0)) {
        cout << "Error: " << sqlite3_errmsg(db) << endl;
    }
}

int ListaEjemplaresPrestados(sqlite3 *db, int clavesocio) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    char resp[10];
    int desplazamiento=0;
    bool ultima;
    bool salir=false;
    int fila=0, i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    if(clavesocio) {
        do {
            cout << "Elegir un prestamo:" << endl << endl;
```

```
            sprintf(consulta, "SELECT claveejemplar,
fecha_prestamo,titulo,formato FROM prestamo NATURAL JOIN
ejemplar NATURAL JOIN libro "
                    "WHERE clavesocio=%d AND
fecha_devolucion IS NULL ORDER BY fecha_prestamo LIMIT 20
OFFSET %d;", clavesocio, desplazamiento);
            rc = sqlite3_prepare_v2(db, consulta, -1,
&ppStmt, NULL);
            if( rc!=SQLITE_OK ){
                cout << "Error: " << sqlite3_errmsg(db) <<
endl;
            } else {
                i = 0;
                while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                    cout << sqlite3_column_int(ppStmt, 0) <<
") " <<
                        sqlite3_column_text(ppStmt, 1) << "
" <<
                        sqlite3_column_text(ppStmt, 2) << "
(" <<
                        sqlite3_column_text(ppStmt, 3) <<
")" << endl;
                    i++;
                }
                sqlite3_finalize(ppStmt);
            }
            ultima = (i < 20);
            while(i < 20) { cout << endl; i++; }
            cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
pagina, (x)salir" << endl;
            cin >> resp;
            switch(resp[0]) {
                case 's':
                    if(!ultima) desplazamiento+=20;
                    break;
                case 'a':
                    if(desplazamiento > 0) desplazamiento-
=20;

                    break;
                case 'x':
                    salir=true;
                    break;
                default:
                    if(isdigit(resp[0])) {
                        fila = atoi(resp);
                        salir=true;
                    }
```

```
                    break;
            }
        } while(!salir);
        return fila;
    } else return 0;
}

void DevolucionPrestamo(sqlite3 *db) {
    int clavesocio;
    int claveejemplar;
    char fecha[12];
    char consulta[1024];
    char notas[256];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira un socio, y se podrá
editar la fecha de devolución y algunas notas de uno de sus
prestamos abiertos." << endl;

    cin.ignore();
    cout << "Socio: ";
    clavesocio = ListaSocios(db);
    if(!clavesocio) return;
    claveejemplar = ListaEjemplaresPrestados(db,
clavesocio);
    if(!claveejemplar) {
        cout << "No hay prestamos abiertos para este socio"
<< endl;
        cin.ignore();
        cin.get();
        return;
    }
    cin.ignore();
    cout << "Fecha de devolucion (AAAA/MM/DD) [dejar en
blanco para fecha actual]: ";
    cin.getline(fecha, 12);
    cout << "Algun comentario: ";
    cin.getline(notas, 256);
    if(strlen(fecha) > 0)
        sprintf(consulta, "UPDATE prestamo SET
fecha_devolucion='%s',notas='%s' "
                "WHERE claveejemplar=%d AND fecha_devolucion
IS NULL;", fecha, notas, claveejemplar);
    else
        sprintf(consulta, "UPDATE prestamo SET
fecha_devolucion=date('now'),notas='%s' "
                "WHERE claveejemplar=%d AND fecha_devolucion
IS NULL;", notas, claveejemplar);
```

```
    if(SQLITE_OK != sqlite3_exec(db, consulta, 0, 0, 0)) {
        cout << "Error: " << sqlite3_errmsg(db) << endl;
        cin.ignore();
        cin.get();
    }
}

int ListaEjemplaresLibro(sqlite3 *db, int clavelibro) {
    sqlite3_stmt *ppStmt;
    int rc;
    char consulta[1024];
    int desplazamiento=0;
    char resp[10];
    bool salir=false;
    bool ultima;
    int fila=0;
    int i;

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Elegir ejemplar" << endl << endl;
        sprintf(consulta, "SELECT
claveejemplar,titulo,numeroorden FROM ejemplar NATURAL JOIN
libro "
                "WHERE clavelibro=%d ORDER BY
titulo,numeroorden LIMIT 20 OFFSET %d;", clavelibro,
desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_int(ppStmt, 0) << ")
" <<
                    sqlite3_column_text(ppStmt, 1) << " ["
<<
                    sqlite3_column_int(ppStmt, 2) << "]" <<
endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(n) editar, (s)ig pagina, (a)nt
```

```
pagina, (x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
            default:
                if(isdigit(resp[0])) {
                    fila = atoi(resp);
                    salir=true;
                }
                break;
        }
    } while(!salir);

    return fila;
}

void HistorialLibro(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    int clavelibro;
    int claveejemplar;
    char consulta[1024];
    int i, desplazamiento=0;
    bool ultima;
    bool salir=false;
    char resp[10];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira un libro y un ejemplar
para mostrar su historial de prestamos." << endl;

    cin.ignore();
    cout << "Libro:";
    clavelibro = ListaLibros(db);
    if(!clavelibro) return;
    claveejemplar = ListaEjemplaresLibro(db, clavelibro);
    if(!claveejemplar) {
        cout << "No hay ejemplares disponibles de este
libro" << endl;
        cin.ignore();
```

```
        cin.get();
        return;
    }
    cin.ignore();

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Historial de ejemplar" << endl << endl;
        sprintf(consulta, "SELECT
fecha_prestamo,fecha_devolucion,titulo,numeroorden FROM
prestamo NATURAL JOIN ejemplar NATURAL JOIN libro "
                "WHERE claveejemplar=%d ORDER BY
fecha_prestamo LIMIT 20 OFFSET %d;", claveejemplar,
desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_text(ppStmt, 0) << "
~ " <<
                    sqlite3_column_text(ppStmt, 1) << " " <<
                    sqlite3_column_text(ppStmt, 2) << " ["
<<
                    sqlite3_column_int(ppStmt, 3) << "]" <<
endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(s)ig pagina, (a)nt pagina,
(x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
```

```
        }
    } while(!salir);
}

void HistorialSocio(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    int clavesocio;
    char consulta[1024];
    int i, desplazamiento=0;
    bool ultima;
    bool salir=false;
    char resp[10];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "A continuacion se pedira un socio para mostrar
su historial de prestamos." << endl;

    cin.ignore();
    cout << "Socio:";
    clavesocio = ListaSocios(db);
    if(!clavesocio) return;
    cin.ignore();

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
    do {
        cout << "Historial del socio" << endl << endl;
        sprintf(consulta, "SELECT
fecha_prestamo,fecha_devolucion,titulo,numeroorden FROM
prestamo NATURAL JOIN ejemplar NATURAL JOIN libro "
                "WHERE clavesocio=%d ORDER BY fecha_prestamo
LIMIT 20 OFFSET %d;", clavesocio, desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_text(ppStmt, 0) << "
~ ";
                // Verificar si la fecha de devolucion es
null
                if(sqlite3_column_text(ppStmt, 1)) cout <<
sqlite3_column_text(ppStmt, 1);
                else cout << " prestado ";
                cout << " " << sqlite3_column_text(ppStmt,
```

```
2) << " [" <<
                    sqlite3_column_int(ppStmt, 3) << "]" <<
endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(s)ig pagina, (a)nt pagina,
(x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
        }
    } while(!salir);
}

void ConsultaPrestamos(sqlite3 *db) {
    sqlite3_stmt *ppStmt;
    int rc;
    int ndias;
    char consulta[1024];
    char numero[15];
    int i, desplazamiento=0;
    bool ultima;
    bool salir=false;
    char resp[10];

    for(int i= 0; i < 24; i++) cout << endl;
    cout << "Prestamos abiertos mas de un numero de dias."
<< endl;

    cin.ignore();
    cout << "Dias: ";
    cin.getline(numero, 15);
    ndias = atoi(numero);

    // Mostrar una lista, teniendo en cuenta que puede haber
más de las que caben en una pantalla.
```

```
    do {
        cout << "Prestamos abiertos mas de " << ndias << "
dias" << endl << endl;
        sprintf(consulta, "SELECT
fecha_prestamo,titulo,numeroorden,socio,JULIANDAY('now')-
JULIANDAY(fecha_prestamo) AS dias "
                "FROM prestamo NATURAL JOIN ejemplar NATURAL
JOIN libro NATURAL JOIN socio "
                "WHERE fecha_devolucion IS NULL AND
JULIANDAY('now')-JULIANDAY(fecha_prestamo)>%d ORDER BY
fecha_prestamo LIMIT 20 OFFSET %d;", ndias, desplazamiento);
        rc = sqlite3_prepare_v2(db, consulta, -1, &ppStmt,
NULL);
        if( rc!=SQLITE_OK ){
            cout << "Error: " << sqlite3_errmsg(db) << endl;
        } else {
            i = 0;
            while(SQLITE_ROW == sqlite3_step(ppStmt)) {
                cout << sqlite3_column_text(ppStmt, 0) << "
~ ";
                cout << sqlite3_column_text(ppStmt, 1);
                cout << " [" << sqlite3_column_int(ppStmt,
2) << "] ";
                cout << "Socio: " <<
sqlite3_column_text(ppStmt, 3);
                cout << " Dias: " <<
sqlite3_column_int(ppStmt, 4) << endl;
                i++;
            }
            sqlite3_finalize(ppStmt);
        }
        ultima = (i < 20);
        while(i < 20) { cout << endl; i++; }
        cout << "\n" << "(s)ig pagina, (a)nt pagina,
(x)salir" << endl;
        cin >> resp;
        switch(resp[0]) {
            case 's':
                if(!ultima) desplazamiento+=20;
                break;
            case 'a':
                if(desplazamiento > 0) desplazamiento-=20;
                break;
            case 'x':
                salir=true;
                break;
        }
```

```
    } while(!salir);
}
```

Descarga de ejemplos

El SQL de SQLite

SQLite no implementa el lenguaje SQL de una forma estricta. Algunas seneencias no están implementadas, y otras se interpretan de un modo diferente. En este capítulo veremos la sintaxis de SQL según lo interpreta SQLite.

Omisiones

Entre las características que no se incluyen, tenemos las siguientes:

Composición

SQLite implementa la composición *LEFT OUTER JOIN*, pero no las composiciones *RIGHT OUTER JOIN* ni *FULL OUTER JOIN*.

Modificación de estructura de tablas

SQLite sólo soporta las variantes *RENAME TABLE* y *ADD COLUMN* de la sentencia ALTER TABLE.

Otras variantes, como *DROP COLUMN*, *ADD CONSTRAINT*, etc, no existen.

Triggers

Están soportados los del tipo *FOR EACH ROW*, pero no los del timpo *FOR EACH STATEMENT*.

Vistas

Las vistas (VIEWs) en SQLite son sólo de lectura. No se pueden ejecutar las sentencias *DELETE*, *INSERT* o *UPDATE* sobre una vista. Sin embargo, se puede crear un trigger que se active ante un intento de borrar, insertar o actualizar una vista, y hacer lo que se necesite en el cuerpo del trigger.

Privilegios

Dado que SQLite lee y escribe en un fichero de disco normal, los únicos permisos de acceso que se pueden aplicar son los normales de acceso a del sistema operativo. Las sentencias GRANT y REVOKE no están implementadas, ya que no tienen sentido en un motor de base de datos embebido.

Comandos especiales

El programa "sqlite3.exe" para línea de comandos permite usar sentencias SQL de forma interactiva sobre cualquier base de datos. También dispone de algunos comandos específicos para modificar la forma de las salidas de los comandos, para activar o desactivar algunos parámetros, etc. En este apéndice los veremos en detalle:

Copia de seguridad

```
.backup [DB] FILE
```

Realiza una copia de seguridad de la base de datos *DB* en el fichero *FILE*. Si no se especifica una base de datos, se hace una copia de la base de datos "main".

Por ejemplo:

```
sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------
--------------------
0    main             C:\programas\ejemplosqlite\agenda.db
1    temp
sqlite> .backup agenda.db agenda.bak
Error: unknown database agenda.db
sqlite> .backup main agenda.bak
sqlite>
```

Bail

```
.bail ON|OFF
```

Detiene la ejecución después de un error. Por defecto está en OFF.

Para ver cómo funciona este comando tenemos que crear un archivo con comandos SQL que después ejecutaremos. Partamos de este ejemplo, en un fichero con el nombre 'prueba.sql':

```
.echo on
.bail off
create table x(
clave integer,
texto text);
insert into x values(1,"uno");
insert into x values(2,"dos");
inserta into x values(3, "tres");
select * from x;
.quit
```

Si ejecutamos este fichero desde "sqlite3.exe", la salida es la siguiente:

```
C:>sqlite3 <prueba.sql
.bail off
create table x(
clave integer,
texto text);
insert into x values(1,"uno");
insert into x values(2,"dos");
Error: near line 8: near "inserta": syntax error
select * from x;
1|uno
2|dos
.quit
```

Si cambiamos la línea de ".bail" y repetimos la ejecución:

```
.echo on
.bail on
create table x(
clave integer,
texto text);
insert into x values(1,"uno");
```

```
insert into x values(2,"dos");
inserta into x values(3, "tres");
select * from x;
.quit
```

Si ejecutamos este fichero desde "sqlite3.exe", la salida es la siguiente:

```
C:>sqlite3 <prueba.sql
.bail on
create table x(
clave integer,
texto text);
insert into x values(1,"uno");
insert into x values(2,"dos");
Error: near line 8: near "inserta": syntax error
```

De modo que si se encuentra un error, la ejecución se detiene.

Lista de bases de datos

```
.databases
```

Lista los nombres y los ficheros correspondientes de las bases de datos abiertas.

```
sqlite> .databases
.databases
seq  name             file
---  ---------------  ----------------------------------------
--------------------
0    main             C:\programas\ejemplosqlite\agenda.db
1    temp
sqlite>
```

Volcado de tablas

```
.dump [TABLE] ...
```

Hace un volcado de la tabla *TABLE* en formato de texto en forma de consultas SQL. Si no se especifica una tabla, se vuelca la base de datos completa. Se pueden especificar varias tablas:

```
sqlite> .dump usuario
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE usuario(
  claveusuario INT,
  nombre TEXT,
  email TEXT
);
INSERT INTO "usuario"
VALUES(1,'Fulano','fulano@dominio.com');
INSERT INTO "usuario"
VALUES(2,'Mengano','mengano@dominio.com');
COMMIT;
sqlite>
```

La tabla puede ser un patrón del tipo que se usa con el operador *LIKE*.

La salida se puede redirigir a un fichero de texto usando el comando ".output".

```
sqlite> .output database.txt
sqlite> .dump
sqlite> .output stdout
sqlite>
```

Activar o desactivar eco

```
.echo ON|OFF
```

Activa o desactiva el eco de los comandos introducidos:

```
sqlite> SELECT CURRENT_DATE;
2012-04-16
sqlite> .echo on
sqlite> SELECT CURRENT_DATE;
SELECT CURRENT_DATE;
2012-04-16
sqlite>
```

Cerrar el programa

```
.exit
```

Sale del programa en línea de comandos.

Cambia el modo de salida

```
.explain [ON|OFF]
```

Permite cambiar de salida. Si se activa se muestran encabezados y se tabulan los listados generados por las sentencias SELECT. Si no se especifica un parámetro, se asume que es ON:

```
sqlite> .explain on
.explain on
sqlite> INSERT INTO usuario VALUES(15, "Filano",
"filano@dominio.com");
sqlite> SELECT * FROM usuario;
clav  nombre         emai
----  -------------  ----
1     Fulano         fulano@dominio.com
2     Mengano        mengano@dominio.com
15    Filano         filano@dominio.com
sqlite>
```

Encabezados de listas

```
.header(s) ON|OFF
```

Activa o desactiva los encabezados de las salidas de las selecciones:

```
sqlite> SELECT * FROM usuario;
1|Fulano|fulano@dominio.com
2|Mengano|mengano@dominio.com
15|Filano|filano@dominio.com
sqlite>
```

Ayuda

```
.help
```

Muestra el listado de comandos especiles, su sintaxis y su función.

Importar desde fichero

```
.import FILE TABLE
```

Importa datos desde el fichero *FILE* a la tabla *TABLE*.

Supongamos que tenemos un fichero de texto con los datos que queremos insertar en una tabla, cada fila en una fila del fichero, y cada columna separada con el carácter '|'. Los valores de las columnas no están delimitados con comillas:

datos.dat

```
16|Federico|federico@dominio.com
17|Mauricio|mauricio@dominio.com
18|Casimiro|casimiro@dominio.com
```

Para importar el contenido del fichero usaremos el comando .import:

```
sqlite> .import datos.dat usuario
sqlite> SELECT * FROM usuario;
clav  nombre         emai
----  -------------  ----
1     Fulano         fulano@dominio.com
2     Mengano        mengano@dominio.com
15    Filano         filano@dominio.com
16    Federico       federico@dominio.com
17    Mauricio       mauricio@dominio.com
18    Casimiro       casimiro@dominio.com
sqlite>
```

Mostrar índices

```
.indices [TABLE]
```

Muestra los nombres de todos los índices. Si se especifica una tabla, sólo se muestran los índices de la tabla indicada, o de las tablas que se ajusten al patrón, los patrones se crean igual que con el operador LIKE, un '_' indica un caracter y '%' una cadena de 1 ó más caracteres:

```
sqlite> CREATE INDEX nombre on usuario(nombre);
sqlite> .indices
nombre
sqlite> .indices us%
nombre
sqlite>
```

Cargar una librería

```
.load FILE [ENTRY]
```

Permite cargar una librería de extensión de SQLite. Estas librerías se crean en C y permiten definir nuevas funciones que se pueden usar en consultas SQLite.

Archivo de registro

```
.log FILE|off
```

Activa un archivo de registro, si se especifica un nombre de archivo en *FILE*. El fichero puede ser stderr o stdout. Para desactivarlo usar `.log off`.

```
sqlite> .log database.log
sqlite> CREATE INDEX nombre on usuario(nombre);
Error: index nombre already exists
sqlite> .log off
sqlite>
```

El fichero "database.log" queda así:

```
C:\programas\ejemplosqlite>type database.log
(1) index nombre already exists
```

Cambiar el modo de salida

```
.mode MODE [TABLE]
```

Permite seleccionar varios modos de salida, donde *MODE* puede tomar cualquiera de los siguientes valores:

- csv: valores separados por comas
- column: columnas alineadas a la izquierda. Se puede especificar el ancho de cada columna con el comando *.width*

- html: formato HTML para tablas
- insert: sentencias de inserción SQL insert para la tabla indicada
- line: un valor en cada línea
- list: valores delimitados por el separador definido mediante el comando *.separator cadena*
- tabs: valores separados por tabuladores
- tcl: elementos de lista TCL

Por ejemplo, veamos la salida en modo *tabs*:

```
sqlite> .mode tabs
sqlite> SELECT * FROM usuario;
claveusuario    nombre  email
1       Fulano  fulano@dominio.com
2       Mengano mengano@dominio.com
15      Filano  filano@dominio.com
sqlite>
```

Nullvalue

```
.nullvalue STRING
```

Permite especificar la cadena que se mostrará en lugar de los valores NULL.

```
sqlite> INSERT into usuario (claveusuario,nombre) values(20,
"Cucufato");
sqlite> SELECT * FROM usuario WHERE claveusuario=20;
claveusuario,nombre,email
20,Cucufato,
sqlite> .nullvalue (nulo)
sqlite> SELECT * FROM usuario WHERE claveusuario=20;
claveusuario,nombre,email
20,Cucufato,(nulo)
sqlite>
```

Output

```
.output FILENAME
```

Envía la salida al fichero indicado por *FILENAME*. Si se quiere volver a enviar la salida a la pantalla usar "stdout" como nombre de fichero.

Cambiar el prompt

```
.prompt MAIN [CONTINUE]
```

Reemplaza el prompt por defecto. El primer parámetro es el prompt principal, que por defecto es "sqlite> ". El segundo es opcional, y es el prompt que se muestra cuando la sentencia no está completa, por defecto es "...> ".

```
sqlite> .prompt "comando> " "sigue> "
comando> SELECT *
sigue> FROM usuario;
claveusuario,nombre,email
1,Fulano,fulano@dominio.com
...
```

Salir del programa

```
.quit
```

Abandona el programa en línea de comandos.

Ejecutar un fichero SQL

```
.read FILENAME
```

Ejecuta un fichero externo con sentencias SQL.

Restaurar base de datos

```
.restore [DB] FILE
```

Restaura el contenido de una base de datos *DB* o de la base de datos "main" si se omite el parámetro, desde un fichero *FILE* generado mediante *.backup*.

```
sqlite> .restore main agenda.bak
sqlite> SELECT * from usuario;
claveusuario,nombre,email
1,Fulano,fulano@dominio.com
2,Mengano,mengano@dominio.com
sqlite>
```

Esquema de base de datos

```
.schema [TABLE]
```

Muestra las sentencias CREATE usadas para crear las tablas de la base de datos, o las de las tablas especificadas mediante el patrón *TABLE*. El patrón se ajusta a las mismas reglas que el operador *LIKE*.

```
sqlite> .schema usuario
CREATE TABLE usuario(
  claveusuario INT,
  nombre TEXT,
  email TEXT
);
sqlite>
```

Definir separadores

```
.separator STRING
```

Cambia el separador usado por el modo de salida y en el comando *.import*.

```
sqlite> .separator "#~$"
sqlite> .mode list
sqlite> SELECT * FROM usuario;
claveusuario#~$nombre#~$email
1#~$Fulano#~$fulano@dominio.com
2#~$Mengano#~$mengano@dominio.com
sqlite>
```

Valores de opciones

```
.show
```

Muestra los valores actuales de algunas de las opciones de configuración.

```
sqlite> .show
     echo: off
  explain: on
  headers: on
     mode: list
nullvalue: "(nulo)"
   output: stdout
separator: "#~$"
    stats: off
    width: 4 13 4 4 4 13 2 13
sqlite>
```

Activar estadísticas

```
.stats ON|OFF
```

Activa (on) o desactiva (off) las estadísticas de SQLite:

```
sqlite> .stats on
sqlite> INSERT INTO usuario
VALUES(3,"Tulano","tulano@dominio.com");
Memory Used:                         65336 (max 123312)
bytes
Number of Allocations:               113 (max 167)
Number of Pcache Overflow Bytes:     8872 (max 17744) bytes
Number of Scratch Overflow Bytes:    0 (max 0) bytes
Largest Allocation:                  48000 bytes
Largest Pcache Allocation:           1160 bytes
Largest Scratch Allocation:          0 bytes
Lookaside Slots Used:                6 (max 83)
Pager Heap Usage:                    8720 bytes
Schema Heap Usage:                   1904 bytes
Statement Heap/Lookaside Usage:      1648 bytes
Fullscan Steps:                      0
Sort Operations:                     0
Autoindex Inserts:                   0
sqlite>
```

Lista de tablas

```
.tables [TABLE]
```

Lista los nombres de las tablas en todas las bases de datos abiertas. Si se especifica el parámetro *TABLE* se mostrarán únicamente los nombres de las tablas que se ajusten al patrón indicado. El patrón tiene la misma sintaxis que el operador *LIKE*.

```
sqlite> CREATE TEMP TABLE user(
...> clave INTEGER,
...> nombre TEXT);
sqlite> .tables
```

```
telefono   telefono2  user       usuario
sqlite>
```

Definir tiempo límite

```
.timeout MS
```

Permite definir el tiempo máximo durante el que se intentarán abrir las tablas bloquedas, en milisegundos.

Anchura de columnas

```
.width NUM1 NUM2 ...
```

Asigna las anchuras de las columnas en los listados de modo columna:

```
sqlite> .mode column
sqlite> .width 3 25 20
sqlite> SELECT * FROM usuario;
cla  nombre                    email
---  ------------------------  --------------------
1    Fulano                    fulano@dominio.com
2    Mengano                   mengano@dominio.com
3    Tulano                    tulano@dominio.com
sqlite>
```

Medida de tiempos

```
.timer ON|OFF
```

Activa o desactiva la medida de tiempo de CPU:

```
sqlite> SELECT * FROM usuario;
cla  nombre                    email
---  ------------------------  --------------------
1    Fulano                    fulano@dominio.com
2    Mengano                   mengano@dominio.com
3    Tulano                    tulano@dominio.com
CPU Time: user 0.000000 sys 0.000000
sqlite>
sqlite> SELECT * FROM usuario;
cla  nombre                    email
---  ------------------------  --------------------
1    Fulano                    fulano@dominio.com
2    Mengano                   mengano@dominio.com
3    Tulano                    tulano@dominio.com
CPU Time: user 0.000000 sys 0.000000
sqlite>
```

Tabla de contenido

- Ayuda
- Importar desde fichero
- Mostrar índices
- Cargar una librería
- Archivo de registro
- Cambiar el modo de salida
- Nullvalue
- Output
- Cambiar el prompt
- Salir del programa
- Ejecutar un fichero SQL
- Restaurar base de datos
- Esquema de base de datos
- Definir separadores
- Valores de opciones
- Activar estadísticas
- Lista de tablas
- Definir tiempo límite
- Anchura de columnas
- Medida de tiempos