Fundamentos de bases de datos

Práctica 2

David Brenchley Uriol

Javier San Andrés de Pedro

Índice

1. Introducción

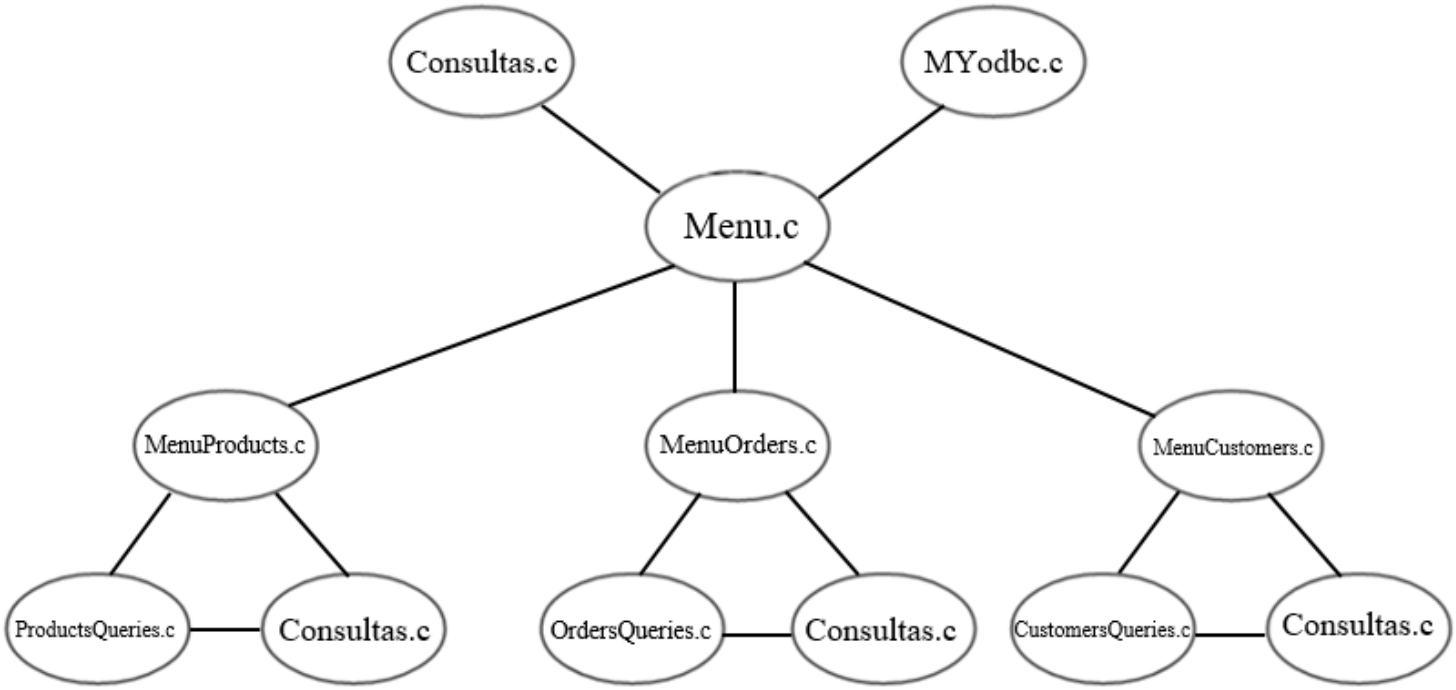
En este apartado, vamos a explicar la modularización realizada para la implementación de la base de datos en C. Nuestro ejecutable de nombre menu, admite o no un parámetro dependiendo de si lo queremos ejecutar en modo de test o en modo diseño (con paginación y otras funcionalidades). Este ejecutable, se sirve de los siguientes ficheros:

- **Menu.c:** contiene el menú principal y se encarga de inicializar la conexión con la base de datos y destruirla. Se comunica a través de diversas funciones con los submenús Products, Orders y Customers mediante sus respectivos ficheros .h ("MenuProducts.h", "MenuOrders.h" y "MenuCustomers.h"). Se comunica con las funciones que gestionan la creación y eliminación de la conexión con la base de datos mediante el fichero "MYodbc.h".
- **MYodbc.c:** contiene la estructura en C de parámetros que se necesitan para comunicarse con la base de datos. Asimismo, también tiene implementadas las funciones de creación de conexión, asignación un objeto que describe y sigue la ejecución de una sentencia SQL, destrucción del objeto anterior, desconexión. Además, las funciones de creación y destrucción de la estructura en C de parámetros al principio citada.
- **MenuProducts.c:** contiene el submenú de productos y las funciones que se encargan de la impresión de los datos obtenidos al realizar las diferentes consultas. Cada función de impresión, se encargará de llamar a la que realiza la consulta correspondiente y está le devolverá el resultado. Después tendrá que imprimirlo con o sin paginación. Para hacer todo esto, se comunica con el fichero "ProductsQueries.h" y con "Consultas.h".
- **MenuOrders.c:** igual que el submenú anterior, misma lógica, pero funciones de impresión aplicadas a consultas diferentes. Se comunica con "OrdersQueries.h" y "Consultas.h"
- **MenuCustomers.c:** igual que los anteriores en nivel lógico. Se comunica con "CustomersQueries.h" y "Consultas.h".
- **Consultas.**c: contiene las funciones que se encargan de recibir y asimilar las opciones de los diferentes menús (según el número que se introduzca se tiene que dirigir a un menú u a otro), función de paginación, función de destrucción del array(tipo char) que contiene las filas de retorno de una consulta, función de reserva de memoria del array anterior, función de reasignación de memoria al array anterior, función de ajuste de datos (para que sea más visual) y una función que ejecuta una consulta de manera general (es decir, cualquier consulta).
- **Consultas.**h: tiene incluida una enumeración que codifica los tipos de datos que se pueden incluir como parámetros o pueden resultar de una selección. También, contiene dos estructuras importantes, la estructura que tiene los posibles parámetros a introducir (ya sean los que se sustituyen en las '?' o los que resultan del select) y la que contiene el resultado de las posibles querys. Además, encapsula las funciones del fichero anterior y sirve de puente para la ejecución de consultas.
- **ProductsQueries.**c: se encarga de realizar las diferentes consultas del submenú Products. Para ello, llama a la función de consulta general (pasando los

argumentos concretos) y, si el retorno puede requerir de paginación, se encarga de crear un array de tipo char y copiar los datos de retorno de la consulta al array (cada fila). El retorno puede ser el array anterior o un retorno concreto. Se encargan de liberar la estructura que resulta de ejecutar la query general.

- **OrdersQueries.**c: igual que la anterior.
- **CustomersQueries.**c: idéntica a las anteriores.

A modo de resumen, presentamos el siguiente esquema (cada enlace representa los ficheros .h que enlazan los archivos .c):



2. Estructura odbc

Para la implementación de la base de datos, hemos diseñado una estructura denominada MyDataBase que contiene tres handles, env, dbc y stmt, y dos retornos ret y ret2, necesarios para conectarse a la base de datos y realizar las diversas queries. Aunque sería posible implementar el menu sin hacer uso de dicha estructura, con el fin de mantener cierta integridad de los handles hemos optado por dicha opción.

Por otro lado, hemos realizado diversos encapsulamientos de las funciones del archivo odbc.c que se nos ha proporcionado. Dichos encapsulamientos, que se encuentran en Myodbc.c, y consisten en la conexión y desconexión a la base de datos, la “allocacion” y liberación del handle stmt para las queries, y finalmente, la inicialización y liberación de la propia estructura MyDataBase.

La inicialización y liberación de la estructura, junto a la conexión y desconexión a la classicmodels se realiza en el menu.c, mientras que la asignación y liberación del stmt se realiza en cada una de las queries.

A continuación, se explica la funcionalidad específica de cada elemento de la estructura:

-SQLHSTMT stmt: Handle que describe y sigue la ejecución de una sentencia SQL, necesaria para poder recibir los datos y avanzar a lo largo de las tablas que devuelven las queries.

-SQLHDBC dbc: Handle que representa la conexión a la base de datos y al driver manager de dicha base, básicamente necesario para enlazarse a la base de datos requerida.

-SQLHENV env: Handle asociado al entorno, al nivel de diagnóstico y los ajustes actuales de cada atributo de dicho entorno

Finalmente, int ret y SQLRETURN ret2 son dos variables encargadas de guardas los retornos de cada una de las funciones asociadas a odbc, para poder realizar el respectivo control de errores y un seguimiento correcto de la ejecución del programa, tanto en el caso de éxito como de fallo.

3. Menu

Como ya hemos comentado en la introducción, el menú principal (situado en el archivo Menu.c) se encarga de crear la conexión con la base de datos, asignar un archivo stmt (citado en el apartado anterior). Si todo va bien, muestra el menú principal que redirecciona a los consiguientes submenús.

Para ello, tiene que pedir un número del 1 al 4. Hemos diseñado una función en Consultas.c que se encarga de realizar esta tarea. Esta función recibe una opción y verifica que se encuentre en uno de los límites establecidos. Si le introducimos parámetros incorrectos, por ejemplo, 250 páginas del Quijote (sin saltos de línea ya que por cada salto de línea se considera que se ha introducido un valor); reproducirá un único mensaje de error donde se volverá a solicitar que se introduzca la opción (esto último dista de otros métodos de leer por teclado cuando se excede el límite de lo que se quiere leer, entre ellos, los que se dan en el menú como referencia).

Una vez se elige una opción correcta, llama a las funciones que se encargan de implementar, mostrar y gestionar los diferentes submenús.

Antes de comentar lo que se hace en cada submenú, me parece importante comentar un poco más a fondo el archivo Consultas.c. La primera función, get_option, es la comentada anteriormente. Por otro lado, la función paginación toma como argumento un array bidimensional de char (donde cada fila contiene una línea de resultados de una consulta) y se encarga precisamente de la paginación de esos resultados (mostrando los resultados de 10 en 10).

La función Destroyer se encarga de liberar el array anteriormente citado. Tiene dos modos: modo 1 en que libera un número determinado de filas y otro en el que libera las filas y el propio array entero. La función Data_alloc se encarga de reservar memoria para dicho array y Data_realloc reasigna la memoria inicialmente reservada.

Data_adjust toma un array de este estilo y una longitud máxima y adecúa los datos en cada fila para que se ajusten a dicha longitud. De esa manera, se consigue que todas las filas acaben en el mismo punto. Esta función solo coloca los primeros caracteres desde el final de fila hasta el primer espacio que se encuentre. Por ello, su funcionamiento y rango de acción es acotado y lo utilizamos específicamente en un apartado concreto.

Por último, vamos a comentar la lógica de la función posiblemente más importante de nuestra práctica: general_query. Está función recibe como parámetros la estructura en C de la base de datos, un puntero a char donde viene el texto de la consulta, un tamaño o número de filas que se encargará de devolver y el número de variables y de retornos que requiere la query. A continuación, se deja que el usuario introduzca parámetros variables. Los primeros, han de corresponder a las variables: hay que especificar, en este orden, tipo de dato y dato en sí. Para los segundos, simplemente hay que especificar el tipo de datos de retorno.

Para codificar los tipos de datos, en su respectivo fichero .h hemos definido una enumeración que distingue entre cuatro tipos diferentes de datos: int, char*, double y date (todos ellos en SQL). Cada uno de ellos, tiene asociado un número (en ese orden números del 0 al 3).

Para guardar las variables que se introducen dentro de los argumentos variables (porque luego vamos a querer bindearlos en nuestro programa), hemos creado una estructura (Parameters) que contiene tres arrays de los diferentes tipos de datos (los de tipo date se guardan como un char) y el número de elementos en cada array. Por cada parámetro que extraigamos, lo guardaremos en cada uno de ellos. Como máximo, se pueden incluir MAX_SELECT parámetros de cada tipo (en nuestro caso 100).

Para los retornos, necesitamos primero tener variables para aplicarles el BindCol. Para ello, podemos servirnos de la estructura anterior y guardarnos cada elemento de cada array en función del tipo de datos que se haya introducido). Ahora bien, estos van a ser los lugares donde se nos almacena cada fila de resultados de la consulta, pero de alguna forma necesitamos poder guardar cada resultado (y no que cambie a cada Fetch).

A fin de conseguir esto, hemos creado la estructura Retornos que contiene tres arrays multidimensionales de cada tipo de data (bidimensional en el caso de int y double y tridimensional en el caso de los strings). En esos arrays es donde vamos a ir almacenando las diferentes filas de resultados. No obstante, existe un número máximo de filas que podemos almacenar, 1000 en nuestro caso; aunque se puede modular con MAX_ROWS.

La función se encarga de reservar memoria para las dos estructuras de Parameters (la de variables y retornos) y devuelve, si todo va bien, un puntero a la estructura Retornos que contiene los resultados de la query.

3.1. Submenús

Por otro lado, el funcionamiento de todos los submenús es exactamente el mismo. Todos tienen una función que se encarga de mostrar el menú correspondiente y solicitar una opción al usuario (utilizando la función get_option de Consultas.c). Después, en función de la elección del usuario, se encargarán de llamar a las funciones que imprimen los resultados de las consultas.

Estas funciones, se encargan de llamar a aquellas que le devuelven los resultados de las consultas en su correspondiente archivo .c (seguir el diagrama inicial) y de la impresión. Tienen dos opciones, o bien imprimen directamente los datos en bruto (para aquellas que no requieren de paginación), o bien delegan la impresión a la función paginación de Consultas.c para aquellas consultas que puedan necesitar potencialmente paginación.

Comentar que todas estas funciones de impresión solo se utilizan en su correspondiente archivo .c, por lo que son de tipo static (no pueden ser usadas fuera de un archivo diferente a ese).

Las funciones que se encargan de realizar las consultas y devolver los resultados están implementadas en los respectivos .c de acuerdo al esquema. Todas estas funciones contendrán la cadena de caracteres con la consulta a ejecutar y se la pasarán como parámetro a la función general_query. Después, se encargarán de recibir el resultado de dicha función y transformarlo de tal forma que se pueda liberar el dato de tipo Retornos que devuelve general_query.

En función si el retorno va a requerir de una posible paginación o no, crearán mediante Data_alloc un array bidimensional de tipo char donde irán guardando cada fila de resultados. Si se quedan sin espacio, llamarán a Data_realloc para reservar más memoria.

Al final, devuelven el resultado de la query sin la estructura de Retornos. Veamos cada submenú.

3.1.1. Products

Tiene cuatro funciones. La primera, la que es utilizada por el menú principal y se encarga de llamar a la segunda función, que muestra el contenido de Products, pedir una opción por teclado y llamar a la tercera y cuarta función que se encargan de imprimir el resultado de cada query.

La primera de las de imprimir resultados de consultas, printStock, llama a la función que ejecuta la consulta en ProductsQueries.c (Stock). La query a ejecutar es la siguiente:

```
SELECT quantityinstock
FROM   products
WHERE  productcode = ?;
```

Y esto es lo que se introduce en la función de consulta general.

```
General_query(pMDB, query, &size, 1, 1, SQLCHAR_C, productcode, SQLINT_C)
```

La segunda es printFind:

```
select p.productcode, p.productname from products p where productname like ?;
```

```
General_query(pMDB, query, size, 1, 2, SQLCHAR_C, likey, SQLCHAR_C, SQLCHAR_C)
```

3.1.2. Orders

Tiene, de manera análoga a Products, cinco funciones donde dos son las del submenú y tres son las de ejecutar consultas.

Entre la de las consultas, la primera es printOpen:

```
select ordernumber from orders where shippeddate is null order by ordernumber;
```

```
General_query(pMDB, query, size, 0, 1, SQLINT_C)
```

La segunda, printRange:

```
select o.ordernumber, o.orderdate, coalesce(cast(o.shippeddate as varchar(10)), 'NULL')
from orders o where orderdate > '2004-03-16' and orderdate < '2005-04-16'
order by ordernumber;
```

```
General_query(pMDB, query, size, 2, 3, SQLCHAR_C, fecha1, SQLCHAR_C, fecha2, SQLINT_C, SQLCHAR_C, SQLCHAR_C)
```

La tercera, printDetails. Esta función ejecuta dos subconsultas. La primera, se encarga de sacar la fecha en que se realizó el pedido, el estado de dicho pedido y el precio que costó:

```
WITH date_status AS (select o.orderdate, o.status, o.ordernumber
                     from orders o where ordernumber = ?),
price as (select sum(o.quantityordered*o.priceeach) as price, o.ordernumber
            from orderdetails o where ordernumber = ? GROUP BY o.ordernumber)
SELECT orderdate, status, price from date_status natural join price;
```

```
General_query(pMDB, query, size, 2, 3, SQLINT_C, ordernumber, SQLINT_C, ordernumber,SQLCHAR_C, SQLCHAR_C, SQLDOUBLE_C)
```

La segunda subconsulta, se encarga de devolver la lista de productos asociado a un determinado ordernumber:

```
select o.productcode, o.quantityordered, o.priceeach
from orderdetails o  where o.ordernumber = ? order by orderlinenumber;
```

```
General_query(pMDB, query, size, 1, 3, SQLINT_C, ordernumber, SQLCHAR_C, SQLINT_C, SQLDOUBLE_C)
```

Utilizamos una consulta auxiliar para saber si el ordernumber existe porque podría ocurrir que un ordernumber no tuviera productos asociados (en el plano lógico no tiene sentido, pero puede haber un error en la base de datos).

3.1.3. Customers

Es prácticamente idéntica a la de Orders, tiene cinco funciones donde dos de ellas se centran en la parte del menú y las otras tres en la realización de las consultas:

La primera, printFind_C:

```
select c.customernumber, c.customername, c.contactfirstname, c.contactlastname
from customers c where c.contactfirstname like ? or c.contactlastname like ?
order by c.customernumber;
```

```
General_query(pMDB, query, size, 2, 4, SQLCHAR_C, contactname, SQLCHAR_C, contactname, SQLINT_C, SQLCHAR_C, SQLCHAR_C, SQLCHAR_C)
```

La segunda, printListProducts:

```
select p.productname, sum(o1.quantityordered) from orderdetails o1, orders o, products p
where o.customernumber=? and o.ordernumber =o1.ordernumber and o1.productcode=p.productcode
group by p.productcode order by p.productcode;
```

```
General_query(pMDB, query, size, 1, 2, SQLINT_C, customernumber, SQLCHAR_C, SQLINT_C)
```

La última, printBalance:

```
with tabla1 as (select sum(pa.amount) as Positivo, pa.customernumber
                from payments pa where pa.customernumber = ?
                group by pa.customernumber),
tabla2 as (select sum(o1.quantityordered*o1.priceeach) as Negativo, o.customernumber
           from orderdetails o1 natural join orders o where o.customernumber= ?
           group by o.customernumber)
select Positivo-Negativo from tabla1 natural join tabla2;
```

```
General_query(pMDB, query, size, 2, 1, SQLINT_C, customernumber, SQLINT_C, customernumber, SQLDOUBLE_C)
```

Ambas dos utilizan una consulta auxiliar para saber si el customernumer existe ya que consideramos que puede darse que un cliente no haya comprado nada todavía.

4. Comprobación

En cuanto a la comprobación de las queries, la principal herramienta que hemos utilizado son los tests, los cual hemos ejecutado, sin paginación, para cada una de las queries. Además, también hemos abierto la base de datos en dbeaver, y hemos cotejado los resultados obtenidos a través de la implementación del menu, con los obtenidas al ejecutar directamente la query en dbeaver y en la terminal psql de postgres. Finalmente, también hemos modificado parcialmente los datos de la base de datos y hemos vuelto a ejecutar las queries para compararlo con la base de datos modificada, es decir, la misma metodología que la practica anterior, pero cotejándolo con lo recibido en el menú.

En cuanto a los tests, todos los resultados se han ido guardando en el archivo tests.log, con la fecha de ejecución y el resultado correspondiente. A continuación, se muestra una captura del comando del makefile que ejecuta dichos tests y parte del resultado obtenido.

```
tests:
    @echo "Running all the tests>>>>>>>>> $(shell date) "
    @echo "\n\n\n-----------------RUNNING TESTS $(shell date)-----------------\n\n\n" >> tests.log
    @make -s test_products
    @make -s test_orders
    @make -s test_customers
```

En el makefile a su vez hay otros 3 comandos que ejecutan los tests para cada menu.

La siguiente imagen muestra el resultado de products_stock

```
Products Menu:
  1.Stock
  2.Find
  3.Back

Enter a number that corresponds to your choice > 1

-------------------------------------------------
Enter productcode > S10_1678
Quantityinstock = 7933
-------------------------------------------------


Products Menu:
  1.Stock
  2.Find
  3.Back

Enter a number that corresponds to your choice >
--------------------------------OK

```

5. Makefile

El *makefile* ha sido realizado con el objetivo de simplificar la compilación y la ejecución de los programas. Para ello, se ha divido en 3 partes distintas, la primera enfocada a crear y poblar la base de datos, la segunda dedicada a la compilación y ejecución del propio menú en C, y la tercera dedicada a la comprobación de los tests proporcionados.

Comandos:

make all: Crea y puebla la base de datos.

make compile: Compila los .c del *menu*, creando los respectivos .o y finalmente el propio archivo *menu*

make menu_design : Ejecuta el menu con paginación y de una forma mucho más estética.

make tests: Ejecuta todos los tests en orden y los guarda en test.log

Importante:

A la hora de ejecutar *menu*, existen **2** posibles ejecuciones: La primera, que se ejecuta simplemente con *./menu*, no incluye paginación y se ha implementado enfocado a los "tests" *.sh*, los cuales se pueden ejecutar directamente sobre él. La segunda, se ejecuta añadiendo un parámetro extra *./menu 1*, por lo que hemos incluido en el *makefile* un comando *menu_design* que lo ejecuta automáticamente.

6. Splint

A la hora de analizar el resultado del comando *splint -nullpass *.c *.h*, se ha de tener en cuenta una serie de factores:

Para empezar, splint considera un error pasar a la función free() un argumento de manera inconsistente, es decir, que hayas pasado a una función un puntero para ser liberado (usando free()), y que todavía exista una variable que apunte a dicha posición de memoria. Esto es debido a que, al pasar un argumento a una función, se crea una variable local para dicha función, la cual liberas, mientras que la original mantiene su valor y por ente la dirección de memoria. Esto no es un problema, ya que se controla de manera interna y las variables "residual" que mantiene la dirección se cede de usar. Se podría evitar pasando el argumento como referencia, pero es innecesario.

En segundo lugar, splint devuelve un aviso cuando existe la posibilidad de que se guarde un Null en alguna variable, lo que es posible en caso de que falle la reserva de memoria. Sin embargo, en una correcta implementación, esto se controla y se realiza un control de

errores adecuado, como hemos realizado nosotros. También es posible que haya Nulls sin inicializar, pero tampoco es un error, pues solo serían temporales.

Por otro lado, a la hora de inicializar nuestra estructura de datos *MyDataBase*, se reserva memoria para ella y más adelante se conecta con la base de datos mediante *odbc_connect*. Sin embargo, a la hora de liberar, primero se desconecta de la base de datos mediante *odbc_disconnect*, lo que libera los handles *dbc* y *env*, y más tarde liberamos la propia estructura. Esto crea un warning al considerar que puede haber fugas, ya que *dbc* y *env*, que están contenidos en *MyDataBase*, pueden seguir apuntado a memoria, aunque ya los hayamos liberado en *odbc_disconnect*.

Finalmente, tenemos numerosas recomendaciones por parte de splint, que sugiere usar *snprintf* en vez de *sprintf*, para evitar posibles overflows. El único inconveniente es que *snprintf* no es incluido por muchas versiones ISO previas a C90, por lo que la bandera -*asci* no lo reconoce. Se podría evitar compilando sin dicha bandera, pero hemos calculado los tamaños para que nunca se de dicho caso.