Universidad Autónoma de Madrid

Departamento de Informática

Data Structures
Assignment 2

Roberto Marabini Ruiz

Contents

1 Goals

End-users never access a database directly (they do not create SQL requests), but instead interact with the database through a user interface that in turn uses a logical layer (software) that builds the queries and launches them. In this assignment we will create one such user interface in C. In particular, we will use the ODBC library to create a program that will connect to the database `classicmodels` provided in the first assignment. Everything you'll learn in this practice is almost directly applicable to any programming language that has an ODBC, JDBC or similar implementation.

2 Accessing the Database with C

As already mentioned, in this assignment we will use ODBC to create a C program to execute certain operations in the database. ODBC is a library for sending database queries and managing the results.

The main resource for information about ODBC is `http://www.unixodbc.org`. The "Manuals" section in this site contains a brief Programming Manual Tutorial covering compilation, database connection, and retrieval of results from queries.

You can find much more detailed tutorials and examples in the Easysoft tutorials on using ODBC from C, including:

- ODBC from C Tutorial Part 1
- ODBC from C Tutorial Part 2 - Fetching Results
- ODBC Code Samples

Finally, for detailed documentation on OBDC functions, check the ODBC Function Summary

In addition, we provide in *Moodle* the files `odbc.h` and `odbc.c`, which contain functions for connecting, disconnecting and printing errors, as well as a collection of examples using these functions.

3 Program to Implement

Your task is to write a C program satisfying the following requirements:

- Displays a menu with 4 options
  1. Products.
  2. Orders.
  3. Customers.
  4. Exit.
- Allows users to select an option using the keyboard.
- Executes the selected option or exits the program if option 4 (Exit) is selected.

When any of the first three options is chosen a new sub-menu must be displayed. The content of these submenus is described in detail below. Some guidelines on how to implement the menu system are given in sub-section 3.4.

3.1 Submenu Products

The "Products" sub-menu should display three options

1. Stock.
2. Find.
3. Back.

- Stock: The user is asked for a product identifier (`productcode`) and the program displays the number of units in stock of this product.
- Find: The user is asked for the name of the product and a list is returned with all the products whose name contains the supplied string. Each line of the output will contain two strings, `productcode` and `productname`. The output will be ordered by the attribute `productcode`. Note that the output may consist of more than one product.
- Back: return to the previous menu

3.2 Submenu Orders

The "Orders" sub-menu should display four options:

1. Open.
2. Range.
3. Detail.
4. Back.

- Open: List all orders that have not been sent yet (date not assigned to `shippeddate`). The output will display on screen the attribute `ordernumber` and it will be ordered by the attribute `ordernumber`.
- Range: Ask the user for two dates in the format YYYY-MM-DD - YYYY-MM-DD. Return a list with all orders placed between these dates. The output must contain the attributes `ordernumber`, `orderdate` and `shippeddate`. Order the output by `ordernumber`.
- Detail: Ask the user for an order identifier (`ordernumber`). Return a list containing the order details. The first line will show the date in which the order was placed (`orderdate`) and whether it has already been sent (`status`), the second line will show the total cost of the order. Then, in one line per product, the program will show the product code (`productcode`), quantity of units ordered (`quantityordered`) and price per unit (`priceeach`). The output of the lines where the products are listed will be ordered by `orderlinenumber`.
- Back: return to the previous menu.

3.3 Submenu Customers

The "Customers" sub-menu should display three options

1. Find.

2. List Products.
3. Balance.
4. Back.

- Find: Ask the user for a client's contact name. Return a list with all clients whose contact name (`contactfirstname`) or surname (`contactlastname`) contains the supplied string. The list must include the client name (`customername`), contact name (`contactfirstname`) and last name (`contactlastname`) and the client identifier (`customernumber`). Order the output by `customernumber`.
- Products: The user supplies a customer's identifier (`customernumber`), and the program returns a list of all products requested by this client belonging to any of his orders. The output should show one line for each requested product, showing the product name `productname`, and the TOTAL number of units requested. That is, if the same product appears in several orders, a single line must appear with the sum of all the units ordered. Sort the output by `productcode`.
- Balance: The user supplies a customer's identifier (`customernumber`) and the program returns the customer's balance, that is, the sum of the payments made minus the sum of the prices of all the products purchased.
- Back: return to the previous menu

Important: To get a score of 10 you must implement paging in this query. That is, if the number of results is greater than 10, only 10 results will be shown per screen; using the keys ">" and "<" the user must be able to move to the next or previous page if they are available.

3.4 Tips for Implementing the Menus

In *Moodle* you can find the file `menu.c`, showing how to implement a system of menus. `menu.c` allows users to choose between a selection of fairy tales or nursery rhymes.

Once the topic has been selected, it offers a list of stories or songs from which to choose a specific one.

Important: For security reasons any query that incorporates a parameter entered by the user will be done using "prepared statements" (see odbc-example4.c)

3.5 Testing

In *Moodle* you will find a collection of test files with the extension `sh`, with content similar to this:

```
#!/usr/bin/expect -f

set timeout -1
spawn ./menu

expect "Enter␣a␣number␣that␣corresponds␣to␣your␣choice␣>␣"
send -- "1\r"

expect "Enter␣a␣number␣that␣corresponds␣to␣your␣choice␣>␣"
send -- "2\r"

expect "Enter␣a␣number␣that␣corresponds␣to␣your␣choice␣>␣"
send -- "4\r"

expect "Enter␣a␣number␣that␣corresponds␣to␣your␣choice␣>␣"
send -- "3\r"

expect eof
```

This file launches the program called `menu`, waits for the string ``Enter a number that corresponds to your choice > '' to appear on the screen and sends a

number followed by a line break in response. In this example, the process is repeated 4 times sending each time the values 1, 2, 4 and 3 respectively.

These test files will be used to grade your assignment, so it is important that you check that they work. In particular, make sure that the strings searched for (`Enter a number that corresponds to your choice >` in the example) are printed out by your program and the format of your lists matches the format that the test files expect.

4 What to Upload to Moodle

Upload to Moodle a single file in zip format containing a single folder named **EDAT_p2_gX_pY**. Here, X is your group, and Y is your team number. This folder should contain, in addition to the program files, a `Makefile` that can be used to:

1. compile your code (make compile)
2. delete, create and populate de data base (make all)

You are also requested to upload the result of executing the command:

```
splint -nullpass *.c  *.h
```

and we hope that all the problems reported by this utility have been remedied. Save the output of `splint` in a file called `splint.log` and attach this file to the material uploaded to *Moodle*.

While a program done in C could be written in a single text file, any serious project requires the source code to be split into several files to make it manageable. It is also expected that the size of the functions created will be moderate (in number of lines), that each function will be commented, and that they will include the name of their creator.

4.1 What to Upload to Moodle after the Second Week

Upload a single zip file following the rules described above, containing:

1. A `Makefile` and all the files needed to compile a program that implements the Products Menu and passes the tests related to this submenu. IMPORTANT: **DON'T include the** `classicmodels.sql` file.

4.2 What to Upload to Moodle at the End

Upload a single zip file following the rules described above, containing:

1. A `Makefile` and all the files needed to compile a the requested program and run the tests. IMPORTANT: **DON'T include the** `classicmodels.sql` file.
2. The `splint.log` file.
3. A report in pdf format that, for each option implemented in the program, includes a brief explanation of the implemented logic, as well as the SQL commands used.

5 Grading Criteria

5 points if the following criteria are met:

- The code uploaded to *Moodle* can be compiled with the supplied makefile.
- There are at least 5 fully working queries. These queries should work for any

instance of the database. (In this context a query is any of the non-trivial menu options, i.e. those that require access to the database.)

- The test files corresponding to the 5 queries must finish their execution.

[5-7] points if the following criteria are met:

- The criteria listed in the previous paragraphs are fully satisfied.
- Queries are robust. For example, the program must respond adequately when provided with non-existent identifiers or the result of the query is the empty set.
- There are at least 6 fully working queries. These queries should work for any instance of the database.
- The test files corresponding to the 6 queries must finish their execution.

[7-8] points if the following criteria are met:

- The criteria listed in the previous paragraphs are fully satisfied.
- A report has been uploaded to *Moodle*. The report should demonstrate the student understanding of the assignment and should answer the question posed in the assignment.
- There are at least 7 fully working queries. These queries should work for any instance of the database.
- The test files corresponding to the 7 queries must finish their execution.
- The code, both in C and SQL, must be readable and commented.
- The program should be properly structured and distributed in several files.

[8.0-8.9] points if the following criteria are met:

- The criteria listed in the previous paragraphs are fully satisfied.

- All the queries are correct for any instance of the database.
- All test files are able to finish their execution.
- The code, both in C and in SQL must be optimized. For example: (1) two queries should not be made when the same result can be obtained with a single query and (2) the queries should be solved by the database, that is, do not bring all the entries of a table and then implement the search in the C program.

[9.0-10.] points if the following criteria are met:

- The criteria listed in the previous paragraphs are fully satisfied.
- Pagination has been implemented.

Late upload of the zip file with the assignment decreases the final mark by one point per day. Late upload of the partial assignment at the end of week 2 is also penalized with 1 point per day. Failure to use git as a version control system will be penalized with -0.5 points.

A Appendix: Git

We recommend using *Git* as a version control tool, specifically, the implementation of the *GitHub* platform.

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

In this appendix we provide a brief guide to *Git*, but we do not cover all options. In particular, we will not explain working with branches, though it can be an interesting mechanism for the student to know. Full documentation on Git can be found in https://git-scm.com/docs

The first step is to connect to https://github.com/ and register for free on the platform.

When you work with Git, you do it on a repository in which the history of versions of your files is stored. There is a remote repository hosted on the Git server; and a local repository, which is a "copy" of the remote repository in a local directory. Normally you start with a cloned remote repository on the working machine. (After creating the repository in *Github*.) Once downloaded, all history, all branches and all versions are available offline. This involves a split between saving a version of one or more files and uploading it to the original server so that others can get those changes. The first action is called commit, while the act of sending that data to the server is called push. The main commands that can be passed to Git to interact and exchange information between the local and remote repository are shown below:

- `git clone remote_repository_url`
  Download and create a link with the remote repository in the directory where we are

- `git add <resource> ...`
  Modify / create one or more files / directories in the repository. Note that Git forces you to do `add` both for new repository items and for changed files. Only the "added" elements will be part of the commit.

- `git rm <resource> ...`
  Delete one or more resources from the repository.

- `git commit -m "Commit message"`
  Update the local repository with all changes made

- `git status`
  Check the repository status, showing three file lists: modified files, new files, and files to include in the commit.

- `git checkout -- <resource>`
  Replace a local resource with the version of the remote repository.

- `git push`
  Upload the local commits to the remote repository. To deal with possible

conflicts with changes made by other users of the repository, it is recommended to first update the local repository.

- `git pull`
  Update the local repository with the latest versions from the remote repository. Updating the repository may cause conflicts because files included in the update have changed locally. Depending on the type of conflict you may have to edit and correct the conflict manually, leaving the file as it should be after applying all local and remote modifications. Once all conflicts have been resolved, the files in conflict must be added again (git add) and commit.