CSCI222

Exercise 3

Data persistence for C++ applications

6 marks

(must be demonstrated in a laboratory class in week 7, 8, or 9)

Aims:

This exercise illustrates a variety of different data persistence technologies that can be used to store large volumes of data for a C++ application.

Objectives:

On completion of this exercise, students should be able to:

- Use “NoSQL” data stores for complex C++ data structures;
- Use SQL data stores for complex C++ data structures;
Overview

The final task in exercise 2 illustrated how a complex C++ data structure could be serialized and saved in a disk file.

But that approach only works with “small” volumes of data. The entire dataset used by the program was loaded from a disk file, kept in memory, updated in memory, and written back to the disk file.

Exercises undertaken at university never involve significant amounts of data, so memory based approaches are feasible. But they don’t scale. It would not for example be practical for Google to use a memory based address book for their gmail users – there are after all 135 million of them.

Realistic data persistence mechanisms will have to store individual records on disk, simply bringing into memory small subsets of the data.

There are many persistence technologies. The technology appropriate to a particular application depends on how it uses its data. Four approaches are illustrated in this exercise.

– Key/value stores (No-SQL);
– Document stores (No-SQL);
– Embedded SQL client (SQL);
– Client access to relational database (SQL).

In this exercise, you are required to implement an example using first one of the No-SQL technologies, and then one of the SQL-based technologies.

Key/value stores

These are ancient – dating back to the first disk based systems in the early 1960s. In the Unix environment, the dbm library included in the early 1980s Berkeley Unix release has proved very influential; there are numerous current derivatives such as gdbm, ndbm, etc. “memcached” is a more recent variant – it scales to use multiple computer systems for temporary data storage and retrieval; memcached itself is transient (data in memory caches) but the memcacheddb variant persists data to disk. The modern Redis and Voldemort key/value stores can be used in the same limited way, but these systems are actually a lot more sophisticated, with richer functionality.

The simplest key/value systems have a key – generally a string (text characters plus a string length integer count – they aren't null terminated C-strings) – and a value that is a block of bytes (again with a byte count). They work like non-memory resident hash tables. You put data in; the key gets written into some “index” part of the disk data store, the “value” recorded with the key is the disk byte address and length for the actual data block. You get data out by specifying the key and leaving it to the system to find and then retrieve data from disk. You can replace data (replacement data aren't re-written in situ, so their length can change). You can delete a key=>data combination (the system looks after any reorganisation of its private data structures). There may be extra functions that are used to tidy up disk usage after a long sequence of inserts and deletes.

The basic systems do not interpret the data values – the data are treated simply as blocks of bytes to be transferred back and forth (this is where Redis and Voldemort offer additional capabilities, they can operate on data values).

Advantages? Minimal overheads, very fast, and no need for a supporting process running a separate database engine.

Disadvantages? So limited. The basic key/value systems are simply about saving and storing blocks of bytes and their associated identifier keys.

So the usage? You want to save and restore complete records whose identifiers you will know.
Document stores

These have only recently been popularised (21st century technology!). Currently, there are two widely used systems—couchdb and mongodb. They have some similarities to “column stores” such as Google's BigTable or Yahoo's Cassandra data store but constitute a separate branch of the “NoSQL” developments.

What is a “document”?

The use of the term “document” is a little bizarre— they don't mean Word-file, or PDF-document or anything like that. A document is approximately a “Javascript Object” which you will remember from CSCI110 is a hash-map of key=>value pairs. The values in a document store? They can be scalars, so one could have the pairs: “name=>Thomas”, “age=>47”. But values can be arrays, so one can have something like: “hobbies=>[ golf sailing skiing]”; or values can themselves be hash maps, e.g. “phones=> { mobile=>041467893, work=>6129457234, home=>61280894563}”. If you want, you can have a value that is a vector of hash-maps of vectors of hash-maps of scalar values.

Documents? They are arbitrarily complex data collections.

And every document is unique—each will have not merely different values for attributes, they almost always have differing sets of attributes. There are no fixed data schemas.

You will be used to working with relational data base systems. With relational systems, you start by creating a conceptual data model, specifying how each record will have the same attributes. You compose a fixed schema—eventually taking the form of your create table SQL script. Any changes later will be costly. You can modify an existing table by adding or deleting columns—but that's something that you should avoid doing on large relational tables!

When you define a table, you specify a set of attributes (columns) that all records (rows) must have. If a particular record doesn't have a known value for some attribute, it still has to be allocated a NULL that takes up exactly the same space in the table as recorded on disk.

The value for a cell in a table (column value for specific row) will be a scalar—a varchar string, an integer, a float, a date, an enum. You cannot have cells that could contain some arbitrary number of scalars—a multi-valued “hobbies” column just doesn't fit with the relational model.

How are data like “hobbies” handled? You know the solution well. There would be a persons table with name, age and other attributes and a primary key identifier. There would be a separate hobbies table with a “foreign key” referencing the persons table. When you want a person’s hobbies, you ask the database engine to join data from persons table and hobbies table.

Documents are far more flexible than relational tables!

There is no fixed schema.

You are storing key=>value pairs; keys ~ column identifiers. If most records (documents) did use the same keys, this key=>value storage would be much more wasteful than a relational table because of the need to store the identifier keys (as strings) for every record (while a relational database has only a single copy of the column names kept in the meta-data for a table, and will use a more efficient approach for allocating space on disk for fixed size data records). But while relational tables will typically have less than a dozen columns with most records having data in each column, documents will typically have a vocabulary of hundreds of “keys” with any individual document possessing values for only a small subset of these keys. (The vocabulary of keys never has to be defined—it just grows as you find the need.)

Multi-valued keys

The multi-valued keys (the ones that have arrays as values or hash-maps as values) add further flexibility. You can easily add or remove the entries in an array like “hobbies”; similarly, you can add new items to a subordinate hash-map like the “phones” in the example earlier.
When you retrieve a document from the data store you get the whole thing – no need for joins, no need for supplementary requests.

If all you need is a few of the “key=>value” pairs from a document, then ask for them – you don't have to retrieve complete documents if you don't need them.

Documents must have unique identifiers – equivalent of database primary keys. They don't support composite primary keys; you can provide a single scalar value if there is a natural key for your documents, or you can have the document store generate unique identifiers (~surrogate primary keys in a relational database).

**Searches**

If you want a document whose unique key you know, it's a single step request – no need for joins to collect data from different tables.

But what about searches for documents where the values for specific keys satisfy some constraints (the equivalent of a relational database system's `select ... where x=? and y=?`)?

Document stores *can* handle such requests, returning a collection of documents (or chosen key=>value subsets from documents) where the requests constraints are satisfied.

By default, searches are full collection searches and so are *desperately slow*.

You can create indexes for a document store, just as you can for relational tables (both document stores and relational database systems will use things like Btree data structures for such indexes). The document store engine will exploit such indexes to speed queries.

While such indexing is possible, it does require planning – you need put more thought into planning your data storage to match frequent usage than you would typically do with a relational database system. (Relational systems support ad hoc queries reasonably well, document systems don't manage as well.) Further, the use of indexes does tend to intrude a little more into the code that you write to access a document store.

**Transactions**

Document stores may provide limited support for transactional operations – atomic: multiple updates that either all complete or none complete, isolated: concurrent operations cannot see partially updated data. But transactional support is less complete than with a proper relational database.

*No joins!*

Many cases that would require “joins” in a relational schema are trivialised because of the use of multi-valued attributes. You get all of a person's phones and hobbies as part of the person document – there is no need for joins here.

But there are cases where a relational engine can use joins to select and retrieve data while a system using a document store will have to do the work in the mainline program. (An example: a person document can have an array containing the document identifiers of his/her children, and may have an entry for “school attended”; school documents have an “address” field. A relational database system would be able to compose a list of “addresses of schools attended by your children” using a single query with a complex join. A program using a document store would have to make several requests to retrieve the same data – get identifiers of children, for each child get school identifier, for each distinct school identifier get address.)

**Updates**

When working with a document store, one typically thinks along the lines

- Retrieve a complete document (converting to instance of program defined class held in memory).
– Update memory copy using mutator functions;
– Send updated version back to document database engine which will replace old version by new version (it may keep both and offer the ability to retrieve earlier versions).

Such an approach is natural for an OO programmer – but represents a costly way of doing an update. If you are just adding a hobby to a person document that has several hundred attributes, it's going to be much more costly to follow this route than simply send a specific update request.

Document stores do allow you to send simple updates (“push 'zumba' into hobbies for person identifier xxx”). The coding of course differs greatly from the get object, update, replace approach. Again, more thought is needed about how you intend to use data than is typical with a relational database (where you would probably not think to go the “retrieve complete record” route!)

**Summary for document stores:**

*Advantages?* Flexibility, adequate support for searches, convenience of getting complete record retrieved in single request.

*Disadvantages?* Need for a separate process running the document database engine; novel database model that may be puzzling to those experienced with relational database systems.

So the usage? *Majority of applications!*

Document stores can handle the requirements of most applications – and often in a more “natural” way than relational systems.

**Embedded SQL engine**

Libraries like sqlite3 offer programs an interface to a simplified form of relational database.

An embedded SQL engine works with code linked into the application that works directly with disk files (much like the key-value systems described earlier). There is no separate database engine.

The relational database is simplified:

– Restricted data types – just “text”, “integer”, “real”.
– No concurrency – so cannot be used in an application that needs to scale up. The library code works directly with the disk file. If you tried for concurrency by having several processes running, each would assume that it had total ownership of the disk file – and any concurrent accesses would simply corrupt the data. If you had a multi-threaded application, you would need to incorporate your own mutual exclusion locks.
– While things like “foreign key constraint” can be defined – they aren't enforced by the underlying engine.

Subject to these simplifications, use of an embedded relational data base engine is generally similar to use of MySQL from PHP as covered in CSCI110.

You start by defining your tables; you use a command line program that creates appropriate file structures for the engine to use when working with these conceptual tables.

Your code uses an API not unlike the mysqli library in PHP. There are functions to open the database, create statements, execute statements and work with result-sets.

**Summary for embedded relational database systems:**

*Advantages?* Familiarity – it's the well known relational model. Light weight – no need for separate processes, relatively small code footprint. Can retrieve data from a single table, or using joins can combine data from many tables.
Disadvantages? Not a scalable solution. “Object-relational” mismatch – if you need to retrieve complete “objects” you will have the usual long winded code unpacking result-set rows and setting fields in an object.

So the usage?

Browsers represent a good example of applications that can exploit embedded relational systems. Browsers are intrinsically single user applications – there aren’t going to be issues of scaling up to handle multiple concurrent users. Browsers need to store a variety of data – cookies from visited sites, recent pages, etc. The records for any particular class of data will all have the same structure (the same information has to be saved – e.g. cookie id, cookie value, host-name, pathname, expiry date); the data elements are mostly simple text.

Many browsers utilize the sqlite3 library!

**Client for database engine**

C++ programs can use client libraries to access data held in commercial scale relational database systems. The arrangements are similar to those met with PHP in CSCI110. Typically, the developer will use a database specific client library – so for Oracle this would be the “oracle call level interface” library, for MySQL the current recommended library is MySQL Connector/C++. (If you want a database neutral interface, there are now implementations of ODBC widely available, or you could use “Embedded SQL” macros with an appropriate macro library. Both these approaches add an extra layer that exposes a standard API that gets mapped down to the database specific drivers.)

The code that gets linked to your application is a client driver. The requests made by your program (e.g. “statement execute query”) get converted into requests that are sent across the network to a database engine where they are executed, a response message is used to return results.

Use of a database client library is generally similar to use of MySQL from PHP as covered in CSCI110.

You start by defining your conceptual data structures, and then the tables. You would probably use the MySQL workbench.

Your code uses an API not unlike the mysqli library in PHP. There are functions to open the database, create statements, execute statements and work with result-sets.

**Summary for client access to relational database system:**


**Disadvantages**? “Object-relational” mismatch – if you need to retrieve complete “objects” you will have the usual long winded code unpacking result-set rows and setting fields in an object. Need for a separate process running the database engine

So the usage?

Most applications can work with relational databases.

**Relational or document?** It's a complex trade-off. One factor is familiarity – most developers have experience with relational systems while document stores are novel. Another factor is the need, or lack of need, for flexibility in the data schema – if your data are uniform and unlikely to vary in nature over the life of your application's usage then the fixed schema of relational tables will suit; if you expect change, documents might be better. If you typically want to retrieve “complete objects” then documents will suit; but if you more usually want to view or update only a small part of the data for an “object” then relational systems will probably be more appropriate.
Tasks

The tasks exploring data persistence technologies are all the essentially the same. Some start with simpler examples, but basically each task involves re-implementing the same two programs. These reuse the MyRecord class from earlier exercises.

The first program will create a number of MyRecord instances and save these to persistent storage. The second program will selectively retrieve records.

Task 1 – You are required to complete either Task 1A (Key/Value) or Task 1B (document store).
Task 2 – You are required to complete either Task 2A (use of embedded sqlite library) or Task 2B (use of MySQL C++ client library).

The “Exercise 1A” part of the description of Task 1A provides details of the two programs that are common to all the tasks – creation of records inserted into the data store, the basic program used to retrieve selected records etc. Details that are standard are not repeated in the descriptions of the other tasks.
No SQL - A) GDBM Key/value storage

The “dbm” family of persistence systems date back to the time when C programs would have used fixed size record structures. If you have a struct like:

```c
struct Fixed {
    char name[30];
    char initials[4];
    // Date of birth - day, month, year
    int day;
    int month;
    int year;
};
```
you can transfer it to/from disk using very fast low-level byte write/read operations something like the following write operation:

```c
struct Fixed astruc;
int strucsize;
int ofile;

strucsize = sizeof(astruc);
astruc.day = 5;
...
ofile=creat(“outfile”,0666);
write(ofile,&astruc,strucsize);
```

If you had an “array” of such structures in a disk file, you could access any specific instance by using “seek” and “tell” operations to position reading and writing.

The “dbm” stores allowed you to have a disk based hash-table structure with a meaningful record key rather than just an array of records that you would have to index by record number. The library functions work with some form of index structure using the keys. When retrieving a value for a key, the dbm library code finds the key in the index; associated with the key there will be a disk position and a byte count for the data. The dbm library code then reads in the data.

The gdbm library has the following functions (there are more “housekeeping” functions):

- `gdbm_open`; creates or re-opens the database file;
- `gdbm_close`; closes the file, flushing any recent changes from memory;
- `gdbm_delete`; delete a record; (reorganises index, marks data space for re-use);
- `gdbm_exists`; checks whether there is a data value with the given key, but does not retrieve data;
- `gdbm_store`; inserts or replaces a record; (can configure so that replacement is disallowed);
- `gdbm_fetch`; retrieve a record given its key;
- `gdbm_firstkey/gdbm_nextkey`; operations that allow an application to retrieve all keys (in essentially random order).

These functions use a struct “datum” (with length and databyte fields) to represent both keys and values. (There tends to be a lot of copying of data. As space for datum structures is allocated dynamically there is considerable scope for memory leaks.)

The example `MyBasicDBMDemo` illustrates simple usage of gdbm. (This example is in the [http://virindi.cs.uow.edu.au/nabg](http://virindi.cs.uow.edu.au/nabg) subversion repository and can simply be downloaded.)
The main() is empty. There is instead a cppUnit test that runs code to illustrate the operations.

The structure of this application is a simple but typical instance of all these “persistence” examples. There are “entity classes” (for data), “boundary classes” (interfaces to users, and to secondary systems like data stores), and there are control classes. In this simple example, the “entity class” is just a little struct defined in FixedStruc.h. The class FixedStrucStore is a “boundary class” - its the interface between the application and the persistent storage. The application code, or in this case the cppUnit tests, represent the “control” element.

The FixedStruc.h header file declares the structure and a function used to create instances:

```c
/* File:    FixedStruc.h
 * Author:  nick
 * Created on 10 December 2012, 10:16 AM */

#ifndef FIXEDSTRUC_H
#define FIXEDSTRUC_H
struct Fixed {
    char name[30];
    char initials[4];
    // Date of birth - day, month, year
    int day;
    int month;
    int year;
};

typedef struct Fixed *FixedStrucPtr;
FixedStrucPtr newStruc(const char* aname, const char* initials, int d, int m, int y);
```

The function newStruc() is defined in the FixedStruc.cpp file:
Actual use of gdbm involves manipulation of lots of little “datum” data structures. It's best to keep details of the persistence mechanism separate from any mainline code – consequently, you almost always see a “boundary” class being defined to handle record storage and retrieval:

```cpp
#include <cstring>
#include "FixedStruc.h"
#include <vector>
#include <gdbm.h>

class FixedStrucStore {
public:

  explicit FixedStrucStore(const char* dbname);

  ~FixedStrucStore();

  // Delete - returns false if there wasn't a record with that key
  bool deleteFixedStruc(const char* key) throw(const char*);

  // Exists - is there a record with a given key
  bool exists(const char* key) throw(const char*);

  // Put - insert/replace a record; returns false if failed
  // (In this example, key is actually the same as name field in data record)
  bool put(const char* key, const FixedStrucPtr data) throw(const char*);

  // Get - get record with key
  FixedStrucPtr get(const char* key) throw(const char*);

  // DB will close in destructor, but can explicitly close it earlier
  void close();

  // The gdbm database actually supports scans to get all keys
  // (many key-value stores don't support this)
  std::vector<const char*> *allKeys();

  bool isOK() { return !(this->invalid); }
  const char* itsName() { return this->name; }

private:

  bool invalid;
  GDBM_FILE db;
  char* name;
  FixedStrucStore(const FixedStrucStore& orig);
  FixedStrucStore& operator=(const FixedStrucStore&);  
};
```
The implementation code involves the calls to the gdbm library functions. (The application has to be linked with gdbm – so the NetBeans project properties must be edited; there is no need for any special include files to be listed, but the linker options must reference the gdbm library:

The implementation for this data access class is:

```cpp
FixedStrucStore::FixedStrucStore(const char* dbname) {
    // Constructors should always succeed. But suppose it failed and
    // the database couldn't be created?
    invalid = true;
    name = strdup(dbname);
    // Open the database file
    // It's to be in read-write mode, and to be created if not already in
    // existence;
    db = gdbm_open(this->name, 512, GDBM_WRCREAT | GDBM_NOLOCK | GDBM_SYNC, OGG4, NULL);
    // Well if attempt to create DB failed, leave the state marked as invalid
    if (db == NULL)
        return;
    // It's ok,
    invalid = false;
}

FixedStrucStore::~FixedStrucStore() {
    if (!invalid) {
        gdbm_close(db);
    }
}

void FixedStrucStore::close() {
    if (!invalid) {
        gdbm_close(db);
    }
    invalid = true;
}
```
bool FixedStrucStore::deleteFixedStruc(const char* key) throw (const char*) {
    if (invalid)
        throw (noDB);
    datum dKey;
    dKey.dptr = strdup(key);
    dKey.dsize = strlen(key) + 1;
    int result = gdbm_delete(db, dKey);
    // Old library, potential for memory leaks!
    // Must manually free memory associated with datum
    free(dKey.dptr);
    return result == 0;
}

bool FixedStrucStore::exists(const char* key) throw (const char*) {
    if (invalid)
        throw (noDB);
    datum dKey;
    dKey.dptr = strdup(key);
    dKey.dsize = strlen(key) + 1;
    int dbRet = gdbm_exists(db, dKey);
    free(dKey.dptr);
    return dbRet != 0;
}

bool FixedStrucStore::put(const char* key, const FixedStrucPtr data) throw (const char*) {
    if (invalid)
        throw (noDB);
    datum dKey;
    datum dVal;
    dKey.dptr = strdup(key);
    dKey.dsize = strlen(key) + 1;
    dVal.dptr = reinterpret_cast<char*>(data);
    dVal.dsize = sizeof(Fixed);
    int dbRet = gdbm_store(db, dKey, dVal, GDBM_REPLACE);
    return (dbRet == 0);
}

std::vector<const char*> *FixedStrucStore::allKeys() {
    if (invalid)
        throw (noDB);
    
    std::vector<const char*> *vptr = new std::vector<const char*>[];
    
    datum akey;
    akey = gdbm_firstkey(db);
    while (akey.dptr != NULL) {
        vptr->push_back(akey.dptr);
        datum nextkey = gdbm_nextkey(db, akey);
        akey = nextkey;
    }
    return vptr;
}

FixedStrucPtr FixedStrucStore::get(const char* key) throw (const char*) {
    if (invalid)
        throw (noDB);
    datum dKey;
    dKey.dptr = strdup(key);
    dKey.dsize = strlen(key) + 1;
    datum dVal;
    dVal = gdbm_fetch(db, dKey);
    free(dKey.dptr);
    return (FixedStrucPtr)dVal.dptr;
}
The unit test code creates instances of FixedStruc, stores them, retrieves them etc:

```cpp
void newtestclass::testPutGet() {
    CPPUNIT_ASSERT((datastore->exists(names[0])));
    FixedStrucPtr pa = newStruc(names[0], initials[0], 5, 12, 1972);
    datastore->put(names[0], pa);
    CPPUNIT_ASSERT(datastore->exists(names[0]));
    FixedStrucPtr pb = datastore->get(names[0]);
    CPPUNIT_ASSERT(pa != NULL);
    CPPUNIT_ASSERT(strcmp(pb->initials, pa->initials) == 0);
    CPPUNIT_ASSERT(pb->year == pa->year);
}

void newtestclass::testPutDeleteGet() {
    populate();
    CPPUNIT_ASSERT(datastore->exists(names[3]));
    FixedStrucPtr pa = datastore->get(names[3]);
    CPPUNIT_ASSERT(pa != NULL);
    CPPUNIT_ASSERT(strcmp(pa->initials, initials[3]) == 0);
    datastore->deleteFixedStruc(names[3]);
    CPPUNIT_ASSERT(!datastore->exists(names[3]));
    FixedStrucPtr pb = datastore->get(names[3]);
    CPPUNIT_ASSERT(pb == NULL);
}
```

If you download the project from the subversion repository it should run:

![image of test result]

**But, but, …**

Of course, by the late 1980s, C programmers were tending to make much more use of dynamically allocated structures, and structs would typically have pointer data members:

```c
struct NotFixed {
    char* name; /* name, arbitrary length, null terminated string from strdup */
    char* initials; /* another strdup allocated string of chars */
    // Date of birth - day, month, year
    int day;
    int month;
    int year;
};
```

Now, with C++, we have things like MyRecord with its data members like vector<string> and map<string, string> that work entirely through pointers to other separately allocated regions of memory.

You cannot simply write a block of bytes if that block contains any pointer (and you shouldn't try to write a block of bytes for a class instance, even if it doesn't have any obvious pointer data members because there are probably hidden elements like “run time type identification”, pointers to virtual tables, etc).

*(Why is there no point trying to write out data blocks that contain pointers? You should know from CSCI124.)*
So what use are key/value systems like gdbm in the modern world?

Extremely useful!

It's simply a matter of your serializing your data into a block of bytes that can be written out and read back in.

How can you serialize complex C++ data structures?

Just carry on using the boost XML serialization code that you have already met!

Exercise 1
Task 1A: Using GDBM Key/value store

This involve two NetBeans projects. The application in the first NetBeans project creates a gdbm database and writes a number of MyRecord records to it. The second application fetches selected MyRecords from this gdbm data base.

The MyException and MyRecord classes are copies of those for the final task in Exercise 2 (with the boost XML serialization function defined in MyRecord). The MyRecordStore class is the same in these two projects.

Project MyRecordKVPersist:

This has a main() that creates a collection of MyRecord instances (using the same createData() function as in previous tasks), and which then saves each individually to the gdbm data store. The “id” field of the record serves as the key when storing that record:
using namespace std;
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <qt4.Qt/qimage.h>
#include <qt4.Qt/qbuffer.h>
#include <qt4.Qt/qbytearray.h>

#include "MyRecord.h"
#include "MyRecordStore.h"

typedef MyRecord* RecordPtr;
vector<RecordPtr> g_theRecords;

string getImage(string filename) {...21 lines }

static void createData() {...189 lines }

int main(int argc, char *argv[]) {
    const char* dbname = "AddressBook.db";
    createData();
    MyRecordStore astore(dbname);
    if (!astoreisOk()) {
        cout << "Failed to open database file" << endl;
        exit(1);
    }
    cout << "Opened database file " << dbname << endl;
    vector<RecordPtr>::const_iterator it;
    for (it=g_theRecords.begin(); it!=g_theRecords.end(); it++) {
        RecordPtr ptr = (*it);
        const char* key = ptr->getID().c_str();
        astore.put(key,ptr);
        cout << "Wrote record " << key << endl;
    }
    return EXIT_SUCCESS;
}

(The complete coding of main.cpp by copying getImage() and createData() from an earlier task.)

The MyRecordStore class is very similar in form to the FixedStrucStore class in the basic example on the subversion server. Really, the only changes are to the put() and get() methods which have different signatures and implementations from those in the earlier example.

bool exists(const char* key) throw (const char*);
// Put - insert/replace a record; returns false if failed
// (In this example, key is actually the same as name field in data record)
bool put(const char* key, const MyRecord *data) throw (const char*);
// Get - get record with key
MyRecord *get(const char*key) throw (const char*);
The implementations of the put() and get() methods must deal with the serialization and deserialization of a complex MyRecord object to and from a block of bytes. This can be done using the boost XML serialization code writing to, or reading from a string buffer. Rather than make the code of get() and put() too complex, private auxiliary member functions have been defined for the class:

```cpp
private:
    bool invalid;
    GBM_FILE db;
    char* name;
    MyRecordStore(const MyRecordStore& orig);
    MyRecordStore& operator=(const MyRecordStore);

    char* recordToChars(const MyRecord *data);
    MyRecord *recordFromFile(char* dataloaded);
};
#endif /* MYRECORDSTORE_H */
```

Methods put() and recordToChars():

```cpp
bool MyRecordStore::put(const char* key, const MyRecord *data) throw (const char*)
{
    if (invalid)
        throw (noDB);

    datum dKey;
    datum dVal;
    dKey.dptr = strdup(key);
    dKey.dsize = strlen(key) + 1;

dVal.dptr = recordToChars(data);
    dVal.dsize = strlen(dVal.dptr) + 1;
    int dBRet = gdbm_store(db, dKey, dVal, GBM_REPLACE);

    free(dKey.dptr);
    free(dVal.dptr);
    return (dBRet == 0);
}
```

```cpp
char* MyRecordStore::recordToChars(const MyRecord *data) {
    std::ostringstream archive_stream;
    boost::archive::xml_archive oa(archive_stream);
    oa & BOOST.Serialization_NVP(data);

    char* chars = strdup(archive_stream.str().c_str());

    return chars;
}
```

Methods get() and recordFromFile():
The project properties for MyRecordKVPersist should specify /usr/include/boost and /usr/include/qt4 as extra include paths for the C++ compiler, and for the linker the libraries libboost_serialization.so, libQtCore.so, libQtGui.so (for the QImage functions), and also the gdbm library:

(You will get a compiler warning about the use of an “anonymous” namespace. Ignore it. The warning relates to an oddity about the way types are defined for gdbm.)
Have you ever had problems trying to remember signatures of functions in standard (and not so standard) libraries?

Have you had to google search for the function definition for some function in a header file that you have included?

NetBeans can help.

Right click on the “include” statement:

```
# if defined (__STDC__) || defined (__cplusplus) || defined(cplusplus)
#define __P(x) x
#else
#define __P(x) ()
#endif

/* External variable, the gdbm build release string. */
extern char *gdbm_version;

/* GDBM C++ support */
#ifdef __cplusplus || defined(cplusplus)
extern "C"
#else
#endif

/* These are the routines! */
extern GDBM_FILE gdbm_open__P((char *, int, int, int, void (*)()));
extern void gdbm_close__P((GDBM_FILE));
extern int gdbm_store__P((GDBM_FILE, datum, datum, int));
extern datum gdbm_fetch__P((GDBM_FILE, datum));
extern int gdbm_delete__P((GDBM_FILE, datum));
extern datum gdbm_firstkey__P((GDBM_FILE));
extern datum gdbm_nextkey__P((GDBM_FILE, datum));
```
Qt Project MyRecordKVView

This project demonstrates that records can be loaded individually from the gdbm database just created.

It uses a Qt form, built using QtBuilder, with a QListView containing the identification keys for the records, a (disabled) QLineEdit for a name, and a QLabel that will be used to hold an image. The list of identifiers will be populated by a call to the allKeys() method of MyRecordStore. A mouse click selection of an entry in the list will result in the program fetching the selected record and displaying its name field, and its image.

The project shares classes MyException, MyRecord, and MyRecordStore with the MyRecordKVPersist project. In addition it has a viewRecordForm class partially generated by QtBuilder, and a MyListModel class (similar to the table model class in Exercise 2). The main() simply builds and runs the graphical interface.

```
#include <QtGui/QApplication>
#include "viewRecordForm.h"

int main(int argc, char * argv[]) {
    // initialize resources, if needed
    Q_INIT_RESOURCE(resfile);

    QApplication app(argc, argv);

    viewRecordForm aform;
    aform.show();

    return app.exec();
}
```

Building the form in QtBuilder:
MyListModel class

Qt has a QStringList class that is suitable for the QListView.

```cpp
#include <QAbstractListModel>
#include <QList>
#include <QStringList>

class MyRecordStore;

class MyListModel : public QAbstractListModel
{
    Q_OBJECT

public:
    MyListModel(QObject *parent = 0);
    void addData(QStringList &data) { namelist = data; }
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;

private:
    QStringList namelist;
};
```

The implementation is straightforward:

```cpp
#include "MyRecordStore.h"
#include <iostream>
MyListModel::MyListModel(QObject *parent)
    : QAbstractListModel(parent) {
}

int MyListModel::rowCount(const QModelIndex & ) const
{
    return namelist.size();
}
QVariant MyListModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (index.row() >= namelist.size() || index.row() < 0)
        return QVariant();

    if (role == Qt::DisplayRole)
        return namelist.at(index.row());
    return QVariant();
}
viewRecordForm class

The program's logic is added to the class generated by the Qt designer.

```cpp
#include "ui_viewRecordForm.h"

class MyListModel;
class MyRecordStore;
class viewRecordForm : public QDialog {
  Q_OBJECT
public:
  viewRecordForm();
  virtual ~viewRecordForm();

public slots:
  void showSelection(const QModelIndex &index);
private:
  Uip::viewRecordForm widget;
  MyListModel *amodel;
  MyRecordStore *dataAccess;
  QStringList ids;
};
```

The constructor opens a connection to the gdbm data store and creates a list-model for the list view. It loads the keys (all the record ids) from the gdbm data store, copying them into QString instances in a QStringList that is added to the list-model. It sets up signal handling so that selection of a key in the list will invoke its showSelection() method.

```cpp
viewRecordForm::viewRecordForm() {
  widget.setupUi(this);

  // Connect to the datastore
  const char* dbname = "AddressBook.db";
  dataAccess = new MyRecordStore(dbname);

  std::vector<string> *names = dataAccess->allKeys();
  std::cout << "Loading list of record ids" << endl;
  std::vector<string>::const_iterator it;
  for (it = names->begin(); it != names->end(); it++) {
    QString qstr((*it).c_str());
    std::cout << (*it).c_str() << endl;
    ids.append(qstr);
  }

  //? leaking memory maybe with those string values in vector names
  amodel = new MyListModel();
  amodel->addData(ids);
  widget.idList->setModel(amodel);

  connect(widget.idList, SIGNAL(clicked(const QModelIndex&)), this,
          SLOT(showSelection(const QModelIndex&)));
}

viewRecordForm::~viewRecordForm() {
  dataAccess->close();
  delete dataAccess;
}
The showSelection() method uses the MyRecordStore object to retrieve a chosen MyRecord from the gdbm database and copy its data into the display fields:

```cpp
void viewRecordForm::showSelection(const QModelIndex &index) {
    QString qstr = ids[index.row()];
    string str = qstr.toStdString();
    MyRecord* selectedRecord = dataAccess->get(str.c_str());
    if (selectedRecord == NULL) {
        cerr << "Failed to retrieve record for id " << str << endl;
        return;
    }
    string personname = selectedRecord->getName();
    widget.nameField->setText(personname.c_str());
    string imagestr = selectedRecord->getImage();
    QByteArray imagebytes = QByteArray::fromBase64(imagestr.c_str());
    QImage img = QImage::fromData(imagebytes);
    widget.imageLabel->setPixmap(QPixmap::fromImage(img));
    delete selectedRecord;
}
```

The compiler options need to mention the boost include files:

```
Build
  Qt
  C++ Compiler
```

and the linker should reference the extra libraries `/usr/lib/x86_64-linux-gnu/libboost_serialization.so` and `/usr/lib/x86_64-linux-gnu/libgdbm.so`.

It should run (you do need to copy the data file created in the 'persist' project):
There is a short introduction to MongoDB on the Internet; and quite a lot of reference information on its home site.

The mongodb database engine should be running as a standard service on the laboratory computers.

The mongo client program can be run from the command line. This client program allows you to quickly check the contents of any document database that you have created; the role of this program is analogous to that of the sqlplus client that you will have used to access Oracle in CSCI235. The “command language” for the mongo client is essentially Javascript (with a few minor modifications and extensions).

When you first connect, you have an empty database engine (default configuration options assign it the name “test”). You can create new databases on “test”, or select a previously existed database, using the “use” command.

We can create a new MongoDB collection (i.e. “table”) by just referencing it in a command. Here, I create a new collection - “persons” - by inserting the couple of records in that collection:

```
> db.persons.insert(
... { _id: 'tom', name: 'Thomas',
... roles: [ 'Manager', 'Accountant' ],
... phones: [ '0466666666', '0282483140' ] } );
> db.persons.insert( 
... { _id: 'dick', name: 'Dick', roles: [ 'Accountant' ] });
```

(I have chosen to provide the “_id” identifier for these records. “_id” is roughly equivalent to the “primary key” in a relational table; mongodb will generate unique identifiers if “_id” is not supplied – a bit like surrogate primary keys for a relational table.)

The different “documents” (i.e. rows) in a collection can vary widely in their “fields” (field ~ column). Fields can be simple scalar types – both my records had scalar type for field “name”; or they can be arrays like the “roles” in my example; or they can be hash-maps like my “phones”.

You can do something equivalent to “select * from persons” - in mongodb that is a find operation on a collection:

```
> db.persons.find();
{ "id" : "tom", "name" : "Thomas", "roles" : [ "Manager", "Accountant" ], "phones" : [ "Mobile" : 04666666666, "Work" : "0282483140" ] }
{ "id" : "dick", "name" : "Dick", "roles" : [ "Accountant" ] }
```

Interactive input of data is error prone; but the mongo client can load a (Javascript) file with commands to create records (load filepathname):
The mongo client's `find()` command can in effect have “where” clauses similar to those in a SQL select statement.

For example, we can ask for the records of persons whose roles include 'first aid officer':

```
> db.persons.find({roles: 'first aid officer'});
```

If we didn't want complete records returned, we could specify the fields in the request (like naming columns in a SQL select statement). Here, I ask for name and phone fields:

```
> db.persons.find({roles: 'first aid officer'}, {name: 1, phones: 1});
```

(Malcolm's record didn't include any phones data – doesn't matter; mongodb reports on the data that it can find.)
The first argument to find() is the “where” clause; the second identifies the fields required (the _id field is included in the response automatically). Both arguments are little Javascript hash-maps with key:value pairs; the “name: 1” entry is basically setting “name” to true in this hash-map, which the mongo shell interprets as a request to include the name field in the response.

The retrieval requests can involve functions defined in the mongo system; if for example you had a numeric valued attribute “age” you could run a request like find({age: {$gt: 25}}) to identify the really old people. There are many on-line tutorials illustrating more complex finds, e.g. “Getting started with Mongodb”.

If the value of an attribute is another map structure, like “phones”, you can use path like names (but these must be in quotes). For example, the following request finds persons with mobile phone numbers recorded ($exists is one of the built in functions in mongodb):

```javascript
> db.persons.find({'phones.Mobile': {$exists: true}});
{ "_id": "tom", "name": "Thomas", "roles": [ "Manager", "Accountant" ], 
phones": { 
"Mobile": 046666666666, "Work": 0282483140 } }
{ "_id": "jill", "name": "Jillian", "roles": [ "Secretary", "First aid officer" ], 
phones": { 
"Mobile": 0446667356340, } } 
```

Records can be update by loading them, changing them, and saving them. Tom no longer works as an accountant, he is the boss; record this -

```javascript
> tom = db.persons.findOne({ _id: 'tom' });
{ "_id": "tom", "name": "Thomas", "roles": [ "Manager", "Accountant" ] } 
> tom.roles = [ 'Boss', 'Manager' ];
> db.persons.save(tom);
```

Can again check on accountants and Tom's new record:

```javascript
> db.persons.find({roles: 'Accountant'});
[ { "_id": "dick", "name": "Dick", "roles": [ "Accountant" ] } ]
[ { "_id": "minnie", "name": "Melissa", "roles": [ "Accountant" ], "hobbies": [ "dance", "movies", "gym" ] } ]
> db.persons.find({ id: 'tom' });
[ { "_id": "tom", "name": "Thomas", "roles": [ "Boss", "Manager" ], "phones": { "Mobile": 046666666666, "Work": 0282483140 } } ]
```

The mongodb engine constructs an index (actually a BTree structure such as you may have studied in CSCI203 or CSCI235) based on the “_id” field; so it can quickly retrieve a record given its id. By default, other find operations (find() and findOne() which simply retrieves the first match) involve “table scans” - the mongodb engine must read in each record and check it. **Obviously, this is going to be unacceptably slow on any reasonable sized data collection.**

You have to define additional indexes. You chose the attributes for which indexes will be constructed based on the expected usage of the data. If for example, your application would frequently need to identify persons with a particular role, you could specify that a (“multikey”) index be created.
The “ensureIndex” function will cause the mongodb engine to create an index if one doesn't already exist.

Indexes can be constructed using multiple attributes, and mongodb supports indexing on (x, y) coordinates (which makes it a good candidate for a data store for a location aware application).

Indexing does get quite complex, and serious use of mongodb requires study of its indexing approaches – there are reference data and tutorials on-line.

The Mongodb engine really is using Javascript objects. On disk, they are stored in some private binary format. Data added to the Javascript objects should be text or numbers. Binary data, e.g. small pictures, can be converted to text (base64 encoding) and saved as the value of an attribute but there is a save binary data option (use a char* pointer to the data and a specified byte count, there are different “binary formats” but the “binary general” usually suffices). Large blocks of binary data, or other data, can be stored in a separate file-system managed by the mongodb engine. (The maximum size of an individual mongo record is 16Mb. If you have larger data elements, you have to store parts in mongo's “GridFS” file-store and record references in the main record.)

**Using mongodb from a C++ application.**

Of course, one doesn't want to work with an interactive shell client – imagine doing all one's Oracle work in sqlplus. The shell client allows for simple experimentation. But really one wants to use the data store from application code.

“Drivers” are provided for PHP, Java, Perl, C, and C++. The access models provided by the drivers do vary a little. For C++, data transfer code works with objects from a BSONObj class defined in the driver library. BSONObj objects are created by adding name-value pairs, and transferred to/from the data store using other functions defined in the library. Typically, the program will use some application defined class that suits its needs and have functions that convert to/from BSONObj implemented in the boundary class that connects to the data store.

(C++ driver: Version 3 of the driver was introduced in Feb 2015 and by default you will be reading the Version 3 documentation. The version installed in the lab is the “legacy-1” version based on release 2.6. Take care to read correct documentation – there is a link to 2.6 documentation in the side panel of the mongo site's main documentation pages.)

The MongoDB data collection “demo.persons” just illustrated using the mongo shell client can be accessed using a simple C++ application – MongoDBDemo (this is in the subversion repository). The NetBeans project again has no mainline, its just the classes and some cppUnit test code. It illustrates some features that are not included in your exercise 1B.

The project is comprised of the following files:
The README.txt file has instructions for running the unit tests. The Javascript file has commands to recreate a standardised set of 13 person records in the MongoDB demo.persons collection.

Class Person is a simple entity class with fields for an identifier; a name; vector collections for roles, hobbies, and favorites; a map<string,string> for phones; and an integer data member (coffeesDrunk).

It's the “boundary” class PersonStore that has all the code involving MongoDB. The operations defined for PersonStore include the obvious get() and put() record methods, along with delete() and
the boolean exists(). The class does implement an allKeys() method; this wouldn't really be appropriate with a typical sized MongoDB collection (you wouldn't want the 135 million keys for the Google Gmail members), but it's ok in a little example.

In addition to these basic methods, class PersonStore has findByRole() and observeCoffeeDrinking() - these methods illustrate things that cannot be done with a key/value store like gdbm.

The findByRole() method is a typical search – find all persons whose roles include a specified role, returning these as a collection (it's kind of equivalent to a SQL select * from persons where _id in select personid from roles where role=...; kind of equivalent, but not joins, no subselects.)

Most often, a C++ program would retrieve a record, e.g. a person, from the database, use it, update it, and then send the updated version back to the database where it would overwrite existing data. But that is costly: send request across network to database for record, record retrieved, record serialised in some db specific way and sent back across network, deserialised, used, reserialised, sent back across network, and used for update. Of course SQL allows for updates of individual fields, no need to transfer complete records. The observeCoffeeDrinking() method is used to illustrate the MongoDB equivalent – a request to update an individual field of a record to be performed entirely in the database engine.

```cpp
class PersonStore {
    public:
        explicit PersonStore(const char* collectionname);
        virtual ~PersonStore();

        // Delete - deleting record with given key.
        // documentation doesn't clarify any failure conditions so
        // treat as no fail.
        void deleteRecord(const char* key) throw (const char*);

        // Exists - is there a record with a given key.
        bool exists(const char* key) throw (const char*);

        // Put - insert/rePLACE a record; returns false if failed
        // in this example, key is actually the same as name field in
        bool put(const char* key, const Person &data) throw (const char*);

        // Get - get record with key
        Person *get(const char* key) throw (const char*);

        // Seems that you cannot really close a standard MongoDB connec
        // void close();

        // Get all keys - ok for toy size database but not for a real one
        std::vector<std::string> *allKeys();

        // Can do searches etc with a MongoDB
        std::vector<Person> *findByRole(const char* role);

        // Can do operations on records in database
        void observeCoffeeDrinking(const char* drinkersname);

        bool isOK() {
            return !(this->invalid);
        }

        const char* itsName() {
            return this->name;
        }

    private:
        bool invalid;

    PersonStore(const PersonStore& orig);

    BSONObj convertToBSON(const Person& data);
    Person *convertToPerson(const BSONObj& data);
};
```
The constructor creates an instance of the `mongo::DBClientConnection` class and then makes the connection; the argument to the connect() method identifies the host and port for the MongoDB process. The `collectionname` argument identifies the collection (table) – it will be `demo.persons`.

```cpp
static const char* connectStr = "localhost:27017";
const char* noDB = "Sorry, but was unable to create database so operation failed";

PersonStore::PersonStore(const char* collectionname) {
    // Constructors should always succeed. But suppose it failed and
    // the database couldn't be created
    invalid = true;
    name = strdup(collectionname);
    // Open a connection to the mongodb process file
    try {
        db = new DBClientConnection();
        db->connect(connectStr);
    } catch (constmongo::DBException &e) {
        return;
    }
    // It's ok
}

PersonStore::~PersonStore() {
    if (!invalid) {
        // Close connection
        // there is no close in the basic connection class (though there
        // is a done() in the scopedconnection variant.
        delete db;
    }
}
```

Most of the methods of classes in the C++ driver require arguments or return values that are instances of `BSONObj`; these are really encoded versions of Javascript objects (key=>value maps). These `BSONObj` objects can be constructed procedurally – create an empty `BSONObj`, add things. But the libraries provide alternatives that are often more convenient including both macros and overloaded operators.

The methods in PersonStore for deleting a record and for testing whether a record exists both use the procedural approach to constructing a `BSONObj`:

```cpp
void PersonStore::deleteRecord(const char* key) throw (const char*) {
    if (!invalid)
        throw (noDB);

    // Create a BSON object with _id set to given key value - approximately
    // equivalent to a where clause in SQL
    BSONObjBuilder b;
    b.append("_id", key);
    BSONObj p = b.obj();
    Query q(p);
    db->remove(this->name, q);
}
```

In `deleteRecord`, the `BSONObj` being created is essentially a SQL where clause; we are deleting record(s) `where _id=xxx`; the `DBClientConnection` class provides the “remove” method. The first argument to `remove()` is the name of the collection (i.e. the “table”) – `demo.persons`. 
The `exists()` function uses the `count()` method in the `DBClientConnection` class; it's basically the same as SQL `select count(*) from persons where _id=xxx`.

```cpp
// Exists - is there a record with a given key

bool PersonStore::exists(const char* key) throw (const char*) {
    if (invalid)
        throw (noDB);
    BSONObjBuilder b;
    b.append("_id", key);
    BSONObj p = b.obj();

    string str = this->name;
    unsigned long long num = db->count(str, p);

    return num == 1;
}
```

(There is a kind of “crib sheet” at the [Mongo web site](http://www.mongodb.org) that shows SQL, mongo shell, and C++ versions for a number of standard operations.)

The `put()` and `get()` methods appear deceptively simple – all the real work is being done in auxiliary functions that convert an instance of the program's Person class to a BSONObj representation (and vice versa). Here, the code is using a macro (`MONGO_QUERY_RESULT`) to construct the BSONObj objects needed to represent where clauses. (You can construct queries by explicitly instantiating a `Query` object, or you can use the `MONGO_QUERY` macro. There are two syntaxes that you can use when constructing `Query` objects – function call and “stream”. Look at the examples in the driver documentation and choose the style that you prefer.)

There is a problem in the `put()` method. The mongo shell has a “save” function that inserts a record if it doesn't already exist, or overwrites a record; but there isn't the same function in the C++ driver so the code must explicitly test to determine whether its doing an insert or an update of a complete record. (Sometime after this was written, an “upsert” argument was added that allows `update()` to be used as an insert or update method.) Almost all these methods (insert, update, `findOne`) have the name of the collection as their first argument.

```cpp
bool PersonStore::put(const char* key, const Person &data) throw (const char*) {
    if (invalid)
        throw (noDB);

    BSONObj bsonperson = convertToBSON(data);

    // The mongo shell has a save command that inserts new objects and updates
    // (replaces) existing objects; there doesn't seem to be an equivalent in
    // C++ driver
    if(this->exists(key)) {
        this->db->update(this->name, MONGO_QUERY("_id" << key), bsonperson);
    }
    else {
        this->db->insert(this->name, bsonperson);
    }
    return true;
}
```
The `findByRole()` method may retrieve many `BSONObj` versions of `Person` objects; these need to be converted in `Person` objects and put in a collection. The “where” clause is again a `BSONObj`; this time it has been constructed using the `BSON()` macro:

```cpp
Person *PersonStore::get(const char* key) throw (const char*) {
    if (invalid)
        throw (noDB);
    Person *p;
    BSONObj b = this->db->findOne(this->name, MONGO_QUERY("_id" << key));
    if(b.isEmpty())
        return NULL;
    p = convertToPerson(b);
    return p;
}
```

The `allKeys()` method wants just the identifiers. It would not be appropriate for the database to be asked to return a collection stored person objects (stored as `BSONObjs`) and for the client code to extract the identifiers. Here we need something equivalent to SQL: `select id from persons;` - no where clause, but something that chooses the fields. It's done here in the `allKeys` method in the call to the `query()` method of the `DBClientConnection`. The 2\textsuperscript{nd} argument in this call is an empty `BSONObj` – it represents an empty where clause. There are then a couple of integer arguments before another `BSONObj` that represents (in Javascript really) the request that only the `_id` field be returned.

```cpp
std::vector<Person*> *PersonStore::findByRole(const char* role) {
    vector<Person*> *records = new vector<Person*>();
    BSONObj search = BSON("roles" << role);
    auto_ptr<DBClientCursor> cursor = this->db->query(this->name, search);
    while (cursor->more()) {
        BSONObj nextone = cursor->next();
        Person* aperson = convertToPerson(nextone);
        records->push_back(aperson);
    }
    return records;
}
```
The “hard work” of mapping the program’s Person class to BSONObjs (Binary Javascript objects) is done in the two private auxiliary functions in the PersonStore class.

Arrays of data have to be explicitly handled using an “array builder” function. Nested (key=>value) maps, like the phones in the example, are built as separate BSONObj instances that are then embedded in the overall BSONObj.

The code below constructs the basic BSONObj with a person’s id, name, and coffee consumption. It then adds other elements as needed. If there are no data, e.g. no hobbies recorded, nothing is added. It’s quite different from building a row in a fixed schema SQL database where missing data have to have NULLs written. The code below adds in arrays for roles, hobbies, favs, and adds in a sub-ordinate BSONObj if there are phone data:

```cpp
std::vector<std::string> *PersonStore::allKeys() {
    // It would NOT be reasonable to try to find all keys for a data store
    // of the size where MongoDB would typically get used. But it suits this
    // simple example
    vector<string> *records = new vector<string>;

    /*
     * send a query to the database.
     *
    Parameters
    dbName      to query, format is <dbname>.<collectname>.<collectname>]*
    query        query to perform on the collection. this is a BSONObj (binary JS)
    nToReturn    n to return (i.e., limit). 0 = unlimited
    nToSkip      start with the nth item
    fieldsToReturn optional template of which fields to select. if unspecif
    queryOptions see options enum at top of this file
    */
    BSONObj fieldsToReturn = BSON("_id" << 1);
    auto_ptr<DBClientCursor> cursor = this->db->query(this->name,
      BSONObj(), 0, 0, &fieldsToReturn);
    while (cursor->more()) {
      BSONObj nextone = cursor->next();
      const char* id = nextone.getStringField("_id");
      records->push_back(id);
    }
    return records;
}
```
The BSONObjBuilder and BSONArrayBuilder classes have the methods for adding in key/value pairs (or array elements); when complete, the result is turned into a BSONObj. (It's analogous to using a Java StringBuffer to build up a long string by appending data, and converting to a String when all data have been added.)

The corresponding convertToPerson() method performs the reverse transformation. It has to interrogate the retrieved BSONObj to determine which fields exist, unpacking those that are present to extract data that are used to fill the vector<string> and map<string,string> in the Person object. (If a field is not present, an empty BSONObj is returned rather than NULL; the eoo() function tests whether an object is empty.) In the case of an embedded sub-ordinate BSONObj (for the phones), the code must retrieve the names of the keys in its key=>value map.
Person *PersonStore::convertToPerson(const BSONObj & data) {

    Person *p = new Person(data.getStringField("_id"));
    p->setName(data.getStringField("name"));
    int coffeecups = data.getIntField("coffees");
    // Value will be INT_MIN if no such numeric record
    // just leave coffesDrunk at zero as set by constructor
    if(coffeecups!=INT_MIN) {
        // coffee consumption is recorded in DB
        p->initCoffee(coffeecups);
    }

    BSONElement roledata = data.getField("roles");
    if (roledata.isNull()) {
        // The record held some role data
        vector<BSONElement> theroles = roledata.Array();
        BOOST_FOREACH(BSONElement be, theroles) {
            p->addRole(be.str());
        }
    }

    BSONElement favdata = data.getField("faus");
    if (favdata.isNull()) {
        vector<BSONElement> favs = favdata.Array();
        BOOST_FOREACH(BSONElement bf, favs) {
            p->addFav(bf.str());
        }
    }

    BSONElement hobbydata = data.getField("hobbies");
    if (hobbydata.isNull()) {
        vector<BSONElement> hobbies = hobbydata.Array();
        BOOST_FOREACH(BSONElement bh, hobbies) {
            p->addHobby(bh.str());
        }
    }

    BSONElement phonedata = data.getField("phones");
    if (phonedata.isNull()) {
        //...
        BSONObj bobj = phonedata.Obj();
        // Pick up the field names ("Mobile", "Work", "Home" etc)
        set<string> thefields;
        bobj.getFieldNames(thefields);

        BOOST_FOREACH(string s, thefields) {
            string val = bobj.getStringField(s.c_str());
            p->addPhone(s, val);
        }
    }
    return p;
}
Of course, the include files must be specified for the compiler and the libraries listed for the linker:

(The mongodb C++ client driver makes extensive use of the boost libraries.)

The project downloaded from the subversion repository should run:
Task 1B: Using MongoDB

This involves two NetBeans projects. The application in the first NetBeans project creates a MongoDB data collection and writes a number of MyRecord records to it. The second application fetches selected MyRecords from this MongoDB database. The projects are almost the same as those in the gdbm version (Exercise 1A); again, they reuse classes such as MyException and MyRecord from earlier exercises.
Project MyRecordMongoPersist:

This has a main() that creates a collection of MyRecord instances (using the same createData() function as in previous tasks), and which then saves each individually to the Mongo data store. The “id” field of the record serves as the key when storing that record:

```cpp
#include <iostream>
#include <fstream>
#include <qt4.Qt/qimage.h>
#include <qt4.Qt/qbuffer.h>
#include <qt4.Qt/qbytearray.h>

#include "MyRecord.h"
#include "MyRecordStore.h"

typedef MyRecord* RecordPtr;
vector<RecordPtr> g_theRecords;

string getImage(string filename) {...21 lines }

static void createData() {...189 lines }

int main(int argc, char *argv[]) {

    createData();
    MyRecordStore astore("demo.myrecords");
    if(!astore.isOK()) {
        cout << "Failed to open database file" << endl;
        exit(1);
    } 
    cout << "Created Mongo data collection demo.myrecords" << endl;
    vector<RecordPtr>::const_iterator it;
    for(it=g_theRecords.begin(); it!=g_theRecords.end(); it++) {
        RecordPtr ptr = (*it);
        const char* key = ptr->getID().c_str();
        astore.put(key,ptr);
        cout << "Wrote record " << key << endl;
    }
    return EXIT_SUCCESS;
}
```

(Complete the coding of main.cpp by copying getImage() and createData() from an earlier task.)

The MyRecordStore class is very similar in form to the PersonStore class in the example on the subversion server. The put() and get methods use MyRecord* rather than Person*. There is no operation equivalent to the observeCoffeeDrinking() method; the find by role is to return simply a vector containing the names of persons in a given role (not complete MyRecord objects):
The function implementations are NOT listed here – you should be able to work them out by analogy to the very similar PersonStore code in the example on the subversion server.

The compiler will need specified includes for qt and mongo; the linker will need the following:

```
Item
```

The program should be easy to make to work and run. One can of course use the mongo command line client to view the records that are created in the mongo data base.
Qt Project MyRecordMongoView

This project demonstrates that records can be loaded individually from the mongo database just created.

It uses a Qt form, built using QtBuilder. This is much the same form as that illustrated for the MyRecordKVView project which had a QListView containing the identification keys for the records, a (disabled) QLineEdit for a name, and a QLabel that will be used to hold an image. It has an additional action button and line edit that allow a user to enter a role and get listed (on cout) the names of persons having that role.

The list of identifiers will be populated by a call to the allKeys() method of MyRecordStore. A mouse click selection of an entry in the list will result in the program fetching the selected record and displaying its name field, and its image.

The project shares classes MyException, MyRecord, and MyRecordStore with the MyRecordMongoPersist project. In addition it has a viewRecordForm class partially generated by QtBuilder, and a MyListModel class. The main() simply builds and runs the graphical interface.

```cpp
#include <QtGui/QApplication>
#include "viewRecordForm.h"

int main(int argc, char *argv[]) {
    // initialize resources, if needed
    // Q_INIT_RESOURCE(resfile);
    
    QApplication app(argc, argv);

    viewRecordForm aform;
    aform.show();

    return app.exec();
}
```

Building the form in QtBuilder:

MyListModel class

This is identical to that illustrated in Exercise 1A.

viewRecordForm class

The program's logic is added to the class generated by the Qt designer. The class is similar to that illustrated for Exercise 1A, but there is an additional “slot” for a method that handles listing of persons with a given role:
The constructor is similar to that for the viewRecordForm in exercise 1A, except it opens the mongo database connection rather than the gdbm version and has to set up an extra signal/slot combination.

```cpp
#include "ui_viewRecordForm.h"
class MyListModel;
class MyRecordStore;
class viewRecordForm : public QDialog {
    Q_OBJECT
public:
    viewRecordForm();
    virtual ~viewRecordForm();

public slots:
    void showSelection(const QModelIndex &index);
    void printPersonsInRole();
private:
    Ui::viewRecordForm *widget;
    MyListModel *amodel;
    MyRecordStore *dataAccess;
    QStringList ids;
};
```

The showSelection() method uses the MyRecordStore object to retrieve a chosen MyRecord from the mongo database and copy its data into the display fields; it is identical to the version previously illustrated. The printPersonsInRole gets the role data entered in the lineEdit element "roleField" - if this is empty, the request is ignored. If a role has been specified, the appropriate function in the dataAccess object is used to retrieve a vector<string> with the names; these are then listed to cout. The coding should be easy for you to handle.
The compiler options need to mention the mongo include files and the linker needs all those libraries (you don't need to specify the qt libraries as this is a Qt project).

It should run:

![Image showing a database interface]

**Task 1 – completion (3 marks)**

Demonstrate *either* your gdbm based data persistence implementation, or *your* mongo based data persistence implementation.
**Another Mongodb C++ example**

This is **not** part of Exercise 3. It is just a larger example of Mongo and C++; it is here simply as a convenient place to have it. There are too few examples out on the Internet showing how the MongoDB C++ API gets used. (Note, this example is for the mongo-cxx legacy driver 2.6 using C++ 98.)

This example illustrates the following:

- Mongo documents referencing other mongo documents (kind of like having foreign keys in a relational database – but the other way around(!), here a mongo document representing a “Movie” is going to have a collection of identifiers of the “File” objects that correspond to supplementary data for the movie while a relational database would have the “File” objects having a foreign key to identify the movie that they belong to).
- Using “GENOID” - the mongodb C++ equivalent of an auto-increment primary key.
- Storing binary data blocks with the C++ mongo driver.
- Updating nested collections (sub-objects) with the mongodb C++ driver.
- Using the mongodb C++ driver to update several fields at the same time.
- Storing files with the GridFS extensions for the mongodb C++ driver.

The example is a tiny contrived movie database. There are three kinds of record - “movie”, “person”, and “file”. Person records exist for actors and movie directors – they just have fields for names and a year of birth; actors can appear in several movies, which is why their data are not embedded in movie records but are held as separate records. The “file” records are to be used for supplementary data – such as text files with reviews of the movie or a plot summary, image files such as posters or actor pictures, and possibly video files with short clips. Video files could be large and cause the overall data held for a movie to exceed 16Mbyte – this is the reason why these supplementary data are held in the GridFS file store. (In this example, Person has essentially a “fixed schema” - there are no variations in fields; the schema for Movie is also fixed – movies differ in how many actors and files they have but all movies have the same fields. So the example is simpler than many real Mongo applications with variable schemas.)

The program has a main() driver that creates a fixed set of Person records, then creates some Movie records and updates these with various supplementary data. It is all hard coded. Of course, a real application would have some interactive GUI interface that allowed a user to enter data, update data, and retrieve data.
My best attempt at showing the relationships amongst Movies, Persons, and Files as an entity diagram (not a very good attempt):


There are three classes – Movie and Person represent these basic entities, DataStorer is a boundary class that does much of the work of transferring data between MongoDB and the application. (Not all requests go via DataStorer, some are handled via calls through the DBClientInterface from within main() and at points in Movie.)
The Person class:

This is a simple entity class holding four data members.

The only significant member functions are fromBSON() and toBSON().

The fromBSON member function extracts data from the string and int fields in the mongo structure, copying data into the instance members.

The toBSON member function is used both when creating a new mongo record, and when updating complete records. The records in the mongo database use “GENOID” identifiers (~auto-increment primary keys). If a new Person object is being stored, a GENOID key must be generated; existing Person objects rebuild these object identifiers from their string representation.
The Movie class

```cpp
void Movie::setDollarData(int yearmade, double estimatedearnings) {
    _dollars = estimatedearnings;
    _year = yearmade;
}

void Movie::setTitle(const std::string &title) {
    _title = title;
}

void Movie::setProducer(const std::string &producer) {
    _producer = producer;
}

void Movie::setDirector(const std::string &directoroid) {
    _directoroid = directoroid;
}

void Movie::createZippedCopyOfFile(const std::string &localfile, const std::string &tempname) {
    // Implementation...
}

void Movie::getTitle() {
    // Implementation...
}

void Movie::getDirectoroid() {
    // Implementation...
}

void Movie::getDollarData() {
    // Implementation...
}
```

```python
# Example
movie = Movie()
movie.setDollarData(2015, 100000000)
movie.setTitle('The Big Lebowski')
movie.setProducer('DARRELL SOLOMON')
movie.setDirector('JOHN TURTURRO')
```

class Movie {
    public:

    Movie();
    ~Movie();
    // -------------------- CREATE TIME --------------------
    void setTitle(const std::string& atitle);
    // Director, Producer, and Actors - to be entered as "aid"s of
    // Person objects
    void setDirector(const std::string& directorid);
    void setProducer(const std::string& producerid);
    // -------------------- LOAD, UPDATE, SAVE --------------------
    void setDollarData(int yearmade, double estimatedearnings);

    static void addAFile(const std::string& movieid,
    const std::string& descriptor, const std::string& localfilename);
    static void retrieveFile(const std::string& fileid,
    const std::string& retrieveToLocalNamed);

    // -------------------- MONGO --------------------
    // Conversion to BSONObj sets the "oid" member
    BSONObj toBSON();
    void fromBSON(const BSONObj mongo::BSONObj &bdata);

    std::string getOid() const;
    void printAll(std::ostream &out) const;
    std::string getFileId(const std::string& ident);
private:

    Movie(const Movie& orig);
    static void createZippedCopyOfFile(const std::string& localfile,
    const std::string& tempname);

    struct bstruct {
        int year;
        double earnings;
    };

    std::string _oid;
    std::string _title;
    std::string _directoroid;

    std::vector<std::string> _actors;
    std::map<std::string, std::string> _files;
    bstruct _dollars;

    // Ratings
    double _rottentomatoes;
    double _imdb;
};

inline void Movie::setTitle(const std::string& atitle) {
A “Movie”:

- Has an identifier (_oid) generated by mongo (GENOID()), a surrogate primary key for a relational database.
- A title
- A director – this will be the string representation of the “oid” of a Person object.
- A STL vector of strings identifying actors – these strings are the string versions of the “oids” for other Person objects.
- A STL map< string, string> for any supplementary files; the first string should be a short text identifier such as “plot”, “review”, “poster-1”, “movieclip-A”, …; the second string is a generated identifier for a file stored in GridFS.
- A couple of double values for ratings (taken from real movie web sites – IMDB and “Rotten Tomatoes”).
- A struct with an integer and a double field – this is a very contrived way of having some data that are transferred to/from mongodb in binary.

The example program (main()) creates Movie records with minimal data – GENOID(), title, director. The Movie record then gets saved to mongodb. It then gets pulled back and the data for the embedded struct are added; it is written back to mongodb. The other data – actors and files – are then added using operations that update the record in the database. These steps illustrate use of GridFS, pushing extra entries into a vector sub-object, updating multiple fields of an object in a single step, and adding entries to a sub-ordinate hash-map.

The fromBSON() and toBSON() member functions are similar to those for class Person. The MongoDB C++ API class BSONObjBuilder has member functions for inserting STL collections such as vector and map. Binary data are assigned a length and then copied as (unsigned) chars.
Add simple data elements.

Add vector of strings.

Add map<string,string>

Add binary data block
void Movie::fromBSON(const mongo::BSONObj &bdata) {
    mongo::BSONElement thing;
    bdata.getObjectID(thing);
    mongo::OID anoid = thing.OID();
    this->_oid = anoid.toString();

    this->_title = bdata.getStringField("title");
    //std::cout << "Retrieving movie " << this->_title << std::endl;
    this->_directoroid = bdata.getStringField("director");

    // Don't appear to be able to extract double values directly
    mongo::BSONElement telem = bdata.getField("rottentomatoes");
    this->_rottentomatoes = telem.Double();
    mongo::BSONElement ielem = bdata.getField("imdb");
    this->_imdb = ielem.Double();

    // Extracting a block of binary data is harder than inserting the data
    mongo::BSONElement structelem = bdata.getField("dollarData");
    int elemLength;
    const char* dataptr = structelem.binData(elemLength);
    memcpy(&this->_dollars, dataptr, elemLength);

    mongo::BSONObjIterator actordata(bdata.getObjectField("actors"));
    while (actordata.more()) {
        this->_actors.push_back(actordata.next().String());
    }

    mongo::BSONObjIterator filedata(bdata.getObjectField("movieFiles"));
    while (filedata.more()) {
        mongo::BSONElement fileentry = filedata.next();
        std::string w1 = fileentry.fieldName();
        std::string w2 = fileentry.String();
        std::pair<std::string, std::string> apair(w1, w2);
        this->_files.insert(apair);
    }
}
There is a member function that prints out details of a movie; this involves retrieving the Person records for the director and any actors:

```cpp
void Movie::printAll(std::ostream &out) const {
    std::cout << "-------------------" << std::endl;
    std::cout << "Movie id " << this->oid << std::endl;
    std::cout << this->title << std::endl;
    if (!this->directorid.empty()) {
        std::cout << "Directed by ";
        Person p;
        p.fromString(pdata);
        std::cout << p.getFullName() << std::endl;
    }
    if (this->actors.size() > 0) {
        std::cout << "Actors include : " << std::endl;
        std::vector<std::string>::const_iterator it;
        for (it = this->actors.begin(); it != this->actors.end(); it++) {
            std::cout << actorid << std::endl;
            Person p;
            p.fromString(pdata);
            std::cout << " " << p.getFullName() << std::endl;
        }
        std::cout << "Year of movie " << this->dollars.year << std::endl;
        std::cout << "Estimated earnings $" << this->dollars.earnings
            << "m" << std::endl;
        std::cout << "Ratings:" << std::endl;
        std::cout << "Rotten Tomatoes : " << this->rottentomatoes << std::endl;
        std::cout << "IMDB : " << this->imdb << std::endl;
        if (this->files.size() > 0) {
            std::cout << "Associated files : " << std::endl;
            for (itm = this->files.begin(); itm != this->files.end(); itm++) {
                std::pair<std::string, std::string> anentry = *itm;
                std::cout << " : " << anentry.second << std::endl;
            }
        }
    }
}
```
The code that uses GridFS for files is actually quite simple. (The example application is creating collections in the “demo” database. The collections are “movies” for the Movie records, “persons” for the People records, and “fs” for files (fs is the default with mongo)).

```cpp
void Movie::addAFilm(const std::string& movieid,
                      const std::string& descript, const std::string &localfilename)
{
// Lots to do
// First, compress the file (using a sub-shell) creating a file in /tmp
// (Using tempnam even though this is deprecated)
// Further, it is possible that same tempnam will be reused - and it is
// being used to provide the id for mongo - why, need to find something
// more sophisticated in a real implementation!
    std::string tempname = tempnam("/tmp", "APKIV");
    Movie::createZippedCopyOfFile(localfilename, tempname);
    //...2 lines
    /*...10 lines */
    mongo::GridFS gfs(mongostore->getConnection(), "demo");
    /*...12 lines */
    mongo::BSONObj result = gfs.storeFile(tempname);
    //...3 lines
    std::string fieldident = "moviefiles.");
    fieldident.append(descript);
    mongo::BSONObj filecommand0 = BSON("$set" << BSON(fieldident << tempname));
    mongostore->runCommand(filecommand0, movieoid, moviecollection);
    // Remember to delete that file created in /tmp
    unlink(tempname.c_str());
}

void Movie::retrieveFile(const std::string& fileid,
                         const std::string &retrievetolocalnamed)
{
    mongo::GridFS gfs(mongostore->getConnection(), "demo");
    mongo::GridFile file = gfs.findFileByName(fileid);
    unsigned long long length = file.write(retrievetolocalnamed);
}
```

The `addAFilm()` member function involves updating a sub-ordinate hash-map:

```cpp
    // The file has been transferred, now put an entry in the main movie record
    // Its collection of files needs a new entry - the description string and
    // the key (OID) for the mongo object created
    std::string fieldident = "moviefiles.");
    fieldident.append(descript);
    mongo::BSONObj filecommand0 = BSON("$set" << BSON(fieldident << tempname));
    mongostore->runCommand(filecommand0, movieoid, moviecollection);
```
**DataStorer class**

An instance of this class is used when setting up the connection to the mongo database, for running commands that insert and retrieve BSONObj instances, and for running arbitrary mongo update commands.

```cpp
#include <string>
#include "mongo/client/dbclient.h"
#include "mongo/util/assert_util.h"

class DataStorer {
public:
    DataStorer(const std::string& ahost,
               const std::string& ahost,
               void store(mongo::BSNObj &anitem,
               const std::string &acollection);
    void update(mongo::BSNObj &item,
               const std::string &acollection);

    void runCommand(mongo::BSNObj &command,
                    const std::string &anoidasstr,
                    const std::string &acollection,
                    bool upsert = false);
    mongo::BSNObj fetchObj(const std::string &acollection,
                            const std::string &anoidasstr);
    auto_ptr<mongo::DBClientCursor> fetchMany(
        const std::string &acollection,
        const mongo::Query &quest);
    mongo::DBClientConnection& getConnection() { return _dbcon; }
private:
    mongo::DBClientConnection _dbcon;
    std::string _mongourl;
    std::string _dbname;
};
```

The code is straightforward and is available in the files that can be downloaded from the subversion repository. An example member function is runCommand():
The function is using the mongo::DBClientConnection class's "getLastError()" member function. This effectively causes the program to wait until mongodb has acknowledge that the data have (at least) been written to its journal.
The driver mainline

```cpp
#include <iostream>
#include "Person.h"
#include "Movie.h"
#include "DataStorer.h"

using namespace std;

const char* personcollection = "persons";
const char* moviecollection = "movies";

DataStorer* mongostore;
vector<string> peopleids;
vector<string> movieids;

static void addPerson(Person* p) {...5 lines }
static void createSomePeople() {...97 lines }
static void addMovie(Movie* m) {...5 lines }

static void addDollarDataToAMovie(const string& movied, int yearmade, double earningsestimate) {...10 lines }
static void addActorToAMovie(const string& movied, const string& actorid) {...9 lines }
static void updateMovieRatings(const string& movied, double tomatoe, double imdb) {...4 lines }
static void createMovies() {...84 lines }
static void finallyRetrieveAFile() {...11 lines }

int main(int argc, char** argv) {
    mongo::client::initialize();
    mongostore = new DataStorer("localhost:27017", "demo");

    createSomePeople();
    createMovies();
    finallyRetrieveAFile();
```
The program first creates a few “Person” records inside mongodb:

```
static void addPerson(Person* p) {
    mongo::BSONObj obj = p->toBSON();
    mongoStore->store(obj, personCollection);
    peopleIds.push_back(p->getOid());
}

static void createSomePeople() {
    Person* p;
    // Some movie directors
    p = new Person();
    p->setFamily("Darabont");
    p->setGiven("Frank");
    p->setYearOfBirth(1959);
    addPerson(p);
    delete p;

    p = new Person();
    p->setFamily("Tarantino");
```

Then it builds and saves a few Movie records:

```
static void createMovies() {
    Movie* m;

    m = new Movie();
    m->setTitle("The ShawShank Redemption");
    m->setDirector(peopleIds.at(0));
    addMovie(m);
    delete m;
    addDollarDataToAMovie(movieIds.at(0), 1995, 100);
    addActorToAMovie(movieIds.at(0),
                     peopleIds.at(5));
    updateMovieRatings(movieIds.at(0),
                       8.2, 9.3);
    Movie::addFile(movieIds.at(0), "Plot", "/ShawshankPlot.txt");
    Movie::addFile(movieIds.at(0), "Poster", "/ShawshankPoster.jpg");
    cout << "Built Shawshank" << endl;

    m = new Movie();
    m->setTitle("Pulp Fiction");
```
Note the construction of a command to update multiple fields in the `updateMovie` function.

The program should (!) run:
End of extra mongodb C++ example
Task 2: SQL

The tasks are the same as in part 1; they involve an application that loads data into the database, and an application that retrieves selected records. It's the same data – a collection of MyRecord instances.

All students should have some previous experience of using SQL databases from inside application code (e.g. PHP code in CSCI110). It's always pretty much the same. The application links with a database driver library. The library will define some structs (or classes) for things like “database connection”, “statement”, “result-set”, “row”; it will also have a quite extensive API with numerous functions for preparing SQL statements, setting parameter values, executing the SQL statements, retrieving results, checking for errors and so forth. Each database library is unique; the functions and structs that they define are similar, but distinct. (You can get ODBC/JDBC drivers – these hide the differences amongst databases with an additional layer of software.) The coding is generally highly stylized and repetitious – an excellent candidate for automatic code-generators (you will use automatic Object-Relational-Mapping (ORM) systems in later subjects).

The MyRecord class can be crudely mapped onto relational tables as follows (diagram produced in MySQL workbench):

(You might prefer a different design with a single table for the three map<string,string> data sets – a single table with an extra column identifying the map appropriate to a particular data row rather that a separate table for the data from each different map.)
Simple MySQL create table statements are as follows (other relational systems will have similar but non-identical variants):

```sql
DROP SCHEMA IF EXISTS `mydb` ;
CREATE SCHEMA IF NOT EXISTS `mydb` DEFAULT CHARACTER SET latin1 COLLATE latin1_swedish_ci ;
USE `mydb` ;

-- Table `mydb`.`MyRecord`
CREATE TABLE IF NOT EXISTS `mydb`.`MyRecord` (   `_id` VARCHAR(10) NOT NULL ,
   `name` VARCHAR(45) NOT NULL ,
   `email` VARCHAR(45) ,
   `info` TEXT NULL ,
   `image` MEDIUMTEXT NULL ,
   PRIMARY KEY (`_id`) )
ENGINE = InnoDB;

-- Table `mydb`.`Roles`
CREATE TABLE IF NOT EXISTS `mydb`.`Roles` (   `idPerson` VARCHAR(10) NOT NULL ,
   `role` VARCHAR(45) NOT NULL ,
   INDEX `fk_Roles_MyRecord` (`idPerson` ASC) ,
   CONSTRAINT `fk_Roles_MyRecord` FOREIGN KEY (`idPerson` )
   REFERENCES `mydb`.`MyRecord` (`_id`) 
   ON DELETE NO ACTION
   ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- Table `mydb`.`Phones`
CREATE TABLE IF NOT EXISTS `mydb`.`Phones` (   `idPerson` VARCHAR(10) NOT NULL ,
   `type` VARCHAR(45) NOT NULL ,
   `number` VARCHAR(45) NOT NULL ,
   INDEX `fk_Phones_MyRecord1` (`idPerson` ASC) ,
   CONSTRAINT `fk_Phones_MyRecord1` FOREIGN KEY (`idPerson` )
   REFERENCES `mydb`.`MyRecord` (`_id`) 
   ON DELETE NO ACTION
   ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- Table `mydb`.`Addresses`
CREATE TABLE IF NOT EXISTS `mydb`.`Addresses` (   `idPerson` VARCHAR(10) NOT NULL ,
   `type` VARCHAR(45) NOT NULL ,
   `address` VARCHAR(45) NOT NULL ,
   INDEX `fk_Addresses_MyRecord1` (`idPerson` ASC) ,
   CONSTRAINT `fk_Addresses_MyRecord1` FOREIGN KEY (`idPerson` )
   REFERENCES `mydb`.`MyRecord` (`_id`) 
   ON DELETE NO ACTION
   ON UPDATE NO ACTION)
ENGINE = InnoDB;
```
-- Table `mydb`.`Other`

```sql
CREATE TABLE IF NOT EXISTS `mydb`.`Other` (  
  `idPerson` VARCHAR(10) NOT NULL ,  
  `key` VARCHAR(45) NOT NULL ,  
  `value` VARCHAR(45) NOT NULL ,  
  INDEX `fk_Other_MyRecord1` (`idPerson` ASC) ,  
  CONSTRAINT `fk_Other_MyRecord1`  
    FOREIGN KEY (`idPerson`)  
    REFERENCES `mydb`.`MyRecord` (`_id`)  
    ON DELETE NO ACTION  
    ON UPDATE NO ACTION)  
ENGINE = InnoDB;
```

It's not a particularly exciting relational scheme, and it's a lot simpler than most that you've worked with in your database subjects.

In principle, the application code is manipulating MyRecord instances – adding/deleting roles, or adding/deleting new “other” key=>value pairs. An extra method has been added to the MyRecord class (deleteRole()) so as to illustrate this feature. With a document store like MongoDB, the updated MyRecord class instance is a replacement document – the entire “document” (binary Javascript object) will always get replaced on disk; it is after all a single entity. With relational databases, it's a bit more complex – specific tables in the relational scheme will have to be changed. (It's more complex, but potentially involves less actual slow memory<=>disk transfers.)

If a MyRecord instance is loaded from data in relational tables and then changed, what updates are needed on the tables?

Some automated ORM systems actually track the changes to an object that has been loaded and will only perform the necessary inserts, updates, and deletes on the persistent data tables. If you aren't prepared to go to the trouble of tracking changes (and it is a lot of trouble) then the simple solution is to handle updates by deleting all existing data from the tables and inserting new data – making it as costly as the MongoDB scheme! (Many database systems can partially automate multiple deletions; they would detect the deletion of a MyRecord row and automatically delete linked rows in the other tables such as the Phones table. However, this feature isn't always supported and the example code here handles the deletions explicitly.)

The primary issue explored in task 2 is how to get at such relational data from inside a C++ application.
Task 2A: SQLite

The SQLite software is installed on the Ubuntu systems in the laboratory. The software includes a function library and a little command line application that can be used to create and manipulate tables directly (something equivalent to Oracle’s sqlplus utility that will have been used by students taking CSCI235).

There are a number of tutorials and references for SQLite on the web, including:

- A tutorial illustrating the sqlite3 command line utility application;
- An introduction to use of sqlite from C/C++
- The API (application programmer interface) for the library

**Project MyRecordSQLitePersist:**

Create a new NetBeans project for the SQLite implementation.

**Define the tables**

This could be done in program code as part of the application that populates the data tables with MyRecord instance data, but usually it is more convenient to have tables created outside of the applications.

The sqlite system accepts “foreign key” constraints in table definitions – but doesn't use them. Its range of data types is not quite as comprehensive as that of the typical database system – with sqlite, you can have integer, real, or text. So one might as well work with simpler table definitions!

Add a “createtables.sql” file to the project:
Use the `sqlite3` command line program (this has some built in help). The command line argument names the database file (will be created if necessary) – here `db0` in the current (i.e. NetBeans project) directory (command is: `sqlite3 db0`):
The sqlite3 interpreter can read in .sql files and execute their commands. Process the createtables.sql file, check the dummy entry (and then delete it).

```sql
sqlite> .read createTables.sql
sqlite> select * from myrecord;
testperson|Test|test@test.com|No info
sqlite> delete from myrecord;
sqlite> .exit
```

The application will as always need classes **MyException** and **MyRecord**, and the main.cpp will need the **createData()** and **getImage()** functions as used previously.

Class **MyRecord** has the extra mutator method:

```cpp
void removeRole(string& oldrole); // Accessor functions
```

It's implementation is simple using functions in the STL `<algorithms>` (remove()) and `<vector>` (erase)
The main.cpp code is the same as for the corresponding application in task 1A and 1B:

```cpp
#include "MyRecord.h"
#include "MyRecordStore.h"

typedef MyRecord* RecordPtr;
vector<RecordPtr> g_theRecords;

static const char* getID(string filename) {...}
static void createData() {...}

int main(int argc, char *argv[]) {

    //createData();
    MyRecordStore astore;
    if(!astore.isOK()) {
        cout << "Failed to open database file" << endl;
        exit(1);
    }

    vector<RecordPtr>::const_iterator it;
    for(it=g_theRecords.begin(); it!=g_theRecords.end(); it++) {
        RecordPtr ptr = (*it);
        const char* key = ptr->getID().c_str();
        astore.put(key,ptr);
        cout << "Wrote record " << key << endl;
    }

    return EXIT_SUCCESS;
}
```

But of course, there is a new version of class MyRecordStore – though its interface is pretty much the same as those illustrated in tasks 1A and 1B.
When using a relational database system, an application will open a connection to the database at startup (getting back a pointer to an instance of some library class) and close it at the end; the pointer to the connection object would be passed as an argument to operations that need database access. This example is slightly simpler; there will be a single instance of the MyRecordStore class and this is the only object that accesses the database so the connection is handled as a data member of the class (sqlite3 *db).

The preparation of SQL statements is relatively costly. The SQL strings are parsed; data structures are created to represent the SQL requests (with members that will take arguments that are later bound to any “?” placeholders in the request). When working with a remote database server engine (such as Oracle or MySQL) construction of prepared statements may involve communication with the database server (sometimes, “prepared statement” structures are sent to the server so that it can pre-compile an optimal query implementation using its indexes).

So, typically, necessary prepared statements would be created once and saved (subsequently, they

```cpp
#include <string>
#include <vector>
#include <sqlite3.h>
#include "MyRecord.h"

class MyRecordStore {
public:
    MyRecordStore();
    -MyRecordStore();
    // Delete - returns false if there wasn't a record with that key
    bool deleteRecord(const char* key) throw (const char*);
    // Exists - is there a record with a given key
    bool exists(const char* key) throw (const char*);
    // Put - insert/replace a record; returns false if failed
    // (In this example, key is actually the same as name field in data record)
    void put(const char* key, const MyRecord *data) throw (const char*);
    // Get - get record with key
    MyRecord *get(const char* key) throw (const char*);

    vector<MyRecord*> *getInRole(const char* role) throw (const char*);

    // DB will close in destructor, but can explicitly close it earlier
    void close();

    std::vector<std::string> *allKeys();

    bool isOK() {
        return !(this->invalid);
    }

private:
    bool invalid;
    sqlite3 *db;

    MyRecordStore(const MyRecordStore& orig);
    MyRecordStore& operator=(const MyRecordStore);

    void recordToTables(const MyRecord *data);
};
```
can be “reset” and reused multiple times). Of course, this would mean that the MyRecord class would need a lot of instance members for these statements. So, just to simplify this illustrative example, the SQL statements are prepared, used, and discarded.

The constructor and destructor for MyRecordStore “opens” and “closes” the database connection (since SQLite is an embedded relational engine, this simply means initialising data structures that belong to the library code).

```cpp
#include "MyRecordStore.h"
#include <cstring>
#include <stdlib.h>
#include <iostream>
using namespace std;

const char* databaseStr = "./db0";
const char* noDB = "Sorry, but was unable to open database so operation failed";
const char* prepareFailed = "Failure when preparing statement";
const char* bindFailed = "Failure when binding arguments";
const char* writeFailed = "Failure when writing to database";

MyRecordStore::MyRecordStore() {
    // Constructors should always succeed. But suppose it failed and
    // the database couldn't be created?
    invalid = true;

    // Open the database file
    int rc = sqlite3_open(databaseStr, &(this->db));

    // Well if attempt to open DB failed, leave the state marked as invalid
    if (db == NULL)
        return;
    // It's ok.
    invalid = false;
}

MyRecordStore::~MyRecordStore() {
    if (!invalid) {
        this->close();
    }
}

void MyRecordStore::close() {
    if (!invalid) {
        sqlite3_close(db);
    }
    invalid = true;
}
```

The function that checks whether there is a record with a given identifier is a relatively simple. It has a SQL string that is used with the library function sqlite3_prepare_v2() to build a prepared statement structure whose address will be stored in the local pointer variable “stmt”. (I have chosen to simply terminate execution if a prepare fails – there isn't anything else really to do). If the statement is successfully prepared, the code binds the string argument. The SQL statement is then executed – sqlite3_step(). The result-set data can be accessed through the “stmt” object – here one requests the value in the returned row (count(*)...). The sqlite3_finalize() method clears out the stmt object.
bool MyRecordStore::exists(const char* key) throw (const char*) {
    if (invalid)
        throw (noDB);
    const char* countrecord = "select count(*) from myrecord where _id=?";
    sqlite3Stmt* stmt;
    int rc;
    rc = sqlite3_prepare_v2(db,
        countrecord, strlen(countrecord),
        &stmt, &unused);
    if (rc != SQLITE_OK) {
        cerr << prepareFailed << endl;
        exit(1);
    }
    rc = sqlite3_bind_text(stmt, 1, key, -1, SQLITE_STATIC);
    if (rc != SQLITE_OK) {
        cerr << bindFailed << endl;
        exit(1);
    }
    rc = sqlite3_step(stmt);
    int count = sqlite3_column_int(stmt, 0);
    sqlite3_finalize(stmt);
    return count == 1;
}

The allKeys() method is again fairly simple. (Note the type casting – the sqlite3 library uses “unsigned char*” for its string type, which is incompatible with STL string etc.) (Most SQL client APIs have an “exec()” function to run a SQL query, and “next” function to work through successive rows in a result-set; sqlite3 uses its “step” function for both purposes.)

std::vector<std::string> *MyRecordStore::allKeys() {
    if (invalid)
        throw (noDB);
    std::vector<std::string> *vptr = new std::vector<std::string>();

    const char* countrecord = "select _id from myrecord";
    const char* unused; // Pointer to unused part of sql string (?)
    sqlite3Stmt* stmt;
    int rc;
    rc = sqlite3_prepare_v2(db,
        countrecord, strlen(countrecord),
        &stmt, &unused);
    if (rc != SQLITE_OK) {
        cerr << prepareFailed << endl;
        exit(1);
    }
    rc = sqlite3_step(stmt);
    while (rc == SQLITE_ROW) {
        const char* id = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 0));
        vptr->push_back(id);
        rc = sqlite3_step(stmt);
    }
    sqlite3_finalize(stmt);
    return vptr;
}

The function that returns a collection of MyRecords where the person has a particular role is slightly more complex. If first must identify the appropriate records, and then must build a vector<MyRecord*>

with complete records loaded. Loading a complete MyRecord requires data
from the MyRecord table and all of the other tables. This operation is handled by MyRecord::get(identifier):
The `put()` operation must check whether it is doing an insert or an update; if it is an update, then the crude expedient of “delete and re-insert” is used rather than try to sort out which tables must be changed and in what ways (some tables might not be changed, others might require additions, some deletions might be needed). Actual transfer of data is handled by a private auxiliary member function (`recordToTables()`):

```cpp
vector<MyRecord*> *MyRecordStore::put(const char* key, const MyRecord *data) throw (const char*) {
    if (invalid)
        throw (noDB);
    // Triesome
    // If it exists - delete it.
    if (this->exists(key)) {
        bool done = this->deleteRecord(key);
    }
    // Insert it
    return this->recordToTables(data);
}
```

There are three lengthy member functions – `get()`, `deleteRecord()`, and `recordToTables()`. None are
listed here in their entirety, you will have to fill in the gaps when you implement this exercise. The 
functions are not complex; just long winded as similar operations must be done on the tables for 
roles, for phones, for addresses, for other data, as well as the main myrecord table.

```cpp
bool MyRecordStore::deleteRecord(const char* key) throw (const char*) {
    if (invalid)
        throw (noDB);
    // explicitly delete subordinate records in phones, addresses, others and roles
    // then delete the myrecord entry
    // recreating the prepared statements for each operation - costly
    const char* deletephones = "delete from phones where personid=?";
    const char* deleteaddress = "delete from addresses where personid=?";
    const char* deletetelephone = "delete from other where personid=?";
    const char* deleteroles = "delete from roles where personid=?";
    const char* deleteother = "delete from myrecord where _id=?";
    const char* unused; // Pointer to unused part of sql string (?)

    sqlite3_stmt* stmt;
    int rc;

    // Phones
    rc = sqlite3_prepare_v2(db, 
                          deletephones, strlen(deletephones),
                          &stmt, &unused);
    if (rc != SQLITE_OK) {
        cerr << prepareFailed << endl;
        exit(1);
    }
    rc = sqlite3_bind_text(stmt, 1, key, -1, SQLITE_STATIC);
    if (rc != SQLITE_OK) {
        cerr << bindFailed << endl;
        exit(1);
    }
    rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);

    // Addresses
    rc = sqlite3_prepare_v2(db, 
                          deleteother, strlen(deleteother),
                          &stmt, &unused);
    if (rc != SQLITE_OK) {
        cerr << prepareFailed << endl;
        exit(1);
    }
    rc = sqlite3_bind_text(stmt, 1, key, -1, SQLITE_STATIC);
    if (rc != SQLITE_OK) {
        cerr << bindFailed << endl;
        exit(1);
    }
    rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);

    return rc == SQLITE_DONE;
```

(Similar code clears out addresses, roles, and others before finally dealing with the record in the 
myrecord table):

```cpp
rc = sqlite3_bind_text(stmt, 1, key, -1, SQLITE_STATIC);
if (rc != SQLITE_OK) {
    cerr << bindFailed << endl;
    exit(1);
}
rc = sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

The get() method and the recordToTables() private function are similar in structure. The get() 
method starts by loading in the data that are in the MyRecord table, then adding data from the other 
tables:
The data from the roles, phones, addresses, and other table are all handled in similar manner:

// Other key value data
rc = sqlite3_prepare_v2(db,
    getother, strlen(getother),
    &stmt, &unused);
if (rc != SQLITE_OK) {
    cErr << prepareFailed << endl;
    exit(1);
}
rc = sqlite3_bind_text(stmt, 1, key, -1, SQLITE_STATIC);
if (rc != SQLITE_OK) {
    cErr << bindFailed << endl;
    exit(1);
}

if (sqlite3_step(stmt) != SQLITE_ROW) {
    // No data with that key
    sqlite3_finalize(stmt);
    return NULL;
}

// Assume just the one row!
string id = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 0));
string name = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
string email = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2));
string info = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 3));
string image = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 4));

MyRecord *arc = new MyRecord(id);
arc->setName(name);
arc->setEmail(email);
arc->setInfo(info);
arc->setImage(image);
sqlite3_finalize(stmt);

The recordToTables() function has a similar structure – put an entry in the myrecord table, and then
as needed add data to the other tables:

```c
void MyRecordstore::recordDataTables(const MyRecord* data) {
    if (invalid)
        throw (noDB);

    // Insert the myrecord data first, then deal with other data tables
    const char* myrecord = "insert into myrecord values( ?, ?, ?, ?, ?, ?)";
    const char* putrole = "insert into roles values(?,?)";
    const char* putphone = "insert into phones values (?, ?, ?)";
    const char* putaddress = "insert into addresses values (?, ?, ?)";
    const char* putother = "insert into other values (?, ?, ?)";
    stmt* stmt = new stmt;

    int rc;
    rc = sqlite3_prepare_v2(db,
        putmyrecord, strlen(putmyrecord),
        &stmt, &unused);
    if (rc != SQLITE_OK) {
        cerr << prepareFailed << endl;
        exit(1);
    }

    /* create table myrecord (id text,
            name text,
            email text,
            image text)
    */

    rc = sqlite3_bind_text(stmt, 1, data->getID().c_str(), -1, SQLITE_STATIC);
    rc = sqlite3_bind_text(stmt, 2, data->getName().c_str(), -1, SQLITE_STATIC);
    rc = sqlite3_bind_text(stmt, 3, data->getEmail().c_str(), -1, SQLITE_STATIC);
    rc = sqlite3_bind_text(stmt, 4, data->getInfo().c_str(), -1, SQLITE_STATIC);
    rc = sqlite3_bind_text(stmt, 5, data->getImage().c_str(), -1, SQLITE_STATIC);

    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
        cerr << writeFailed << endl;
        exit(1);
    }
}
```

Save roles, phones, addresses, and final any other data:
The project's properties need to specify the inclusion of /usr/include/qt4 in the compiler options; the application needs the following link libraries:

```
if (data->getOtherKV().size() > 0) {
    rc = sqlite3_prepare_v2(db,
        putother, strlen(putother),
        6, stmt, &unused);
    if (rc != SQLITE_OK) {
        cerr << prepareFailed << endl;
        exit(1);
    }
    map<string, string>::const_iterator it;
    for (it = data->getOtherKV().begin(); it != data->getOtherKV().end(); it++) {
        string anaddress = it->first;
        string anaddress = it->second;
        sqlite3_reset(stmt);
        rc = sqlite3_bind_text(stmt, 1, data->getID().c_str(), -1, SQLITE_STATIC);
        rc = sqlite3_bind_text(stmt, 2, anaddresstype.c_str(), -1, SQLITE_STATIC);
        rc = sqlite3_bind_text(stmt, 3, anaddress.c_str(), -1, SQLITE_STATIC);
        rc = sqlite3_step(stmt);
        if (rc != SQLITE_DONE) {
            cerr << writeFailed << endl;
            exit(1);
        }
    }
    sqlite3_finalize(stmt);
}
```

The program should run – and you can check it using the sqlite3 console application:
**Project MyRecordSQLiteView:**

Create a new NetBeans Qt project for the SQLite version of the display program. This version has extra controls in the GUI for addition and deletion of roles and for printing a selected record.

Of course, the classes MyException, MyRecord, and MyRecordStore are copied from the MyRecordSQLitePersist project. The driver main.cpp will be essentially identical to that used in Task 1A or 1B; the “list view” model class will be identical to that given earlier; the coding for the view form class will be similar with just the extensions needed for the additional operations.

You need only edit the generated stub viewRecordForm class to add the processing functions and set up the signal/slot links.
#include "ui_viewRecordForm.h"
class MyListModel;
class MyRecordStore;
class MyRecord;
class viewRecordForm : public QDialog {
  
public:
  viewRecordForm();
  virtual ~viewRecordForm();

public slots:
  void showSelection(const QModelIndex &index);
  void printSelectedRecord();
  void addRole();
  void removeRole();

private:
  void printout(MyRecord &data);
  Ui::viewRecordForm *widget;
  MyListModel *amodel;
  MyRecordStore *dataAccess;
  QStringList ids;
  int currentSelection;
  
};

viewRecordForm::viewRecordForm() {
  widget.setupUi(this);
  currentSelection = -1;
  // Connect to the data collection -
  dataAccess = new MyRecordStore();

  std::vector<std::string> *names = dataAccess->allKeys();
  std::cout << "Loading list of record ids" << endl;
  std::vector<std::string>::const_iterator it;
  for (it = names->begin(); it != names->end(); it++) {
    QString qstr(QString(*it).c_str());
    std::cout << (*it).c_str() << endl;
    ids.append(qstr);
  }

  amodel = new MyListModel();
  amodel->addData(ids);
  widget.idList->setModel(amodel);

  connect(widget.idList, SIGNAL(clicked(const QModelIndex&)), this,
          SLOT(showSelection(const QModelIndex&)));

  connect(widget.printButton, SIGNAL(clicked()), this,
          SLOT(printSelectedRecord()));
  connect(widget.addButton, SIGNAL(clicked()), this,
          SLOT(addRole()));
  connect(widget.removeButton, SIGNAL(clicked()), this,
          SLOT(removeRole()));
}

viewRecordForm::~viewRecordForm() {
  delete dataAccess;
}
The showSelection() method that displays an image and name for a chosen record is the same as in earlier examples. A couple more methods are illustrated here, the others you should be able to complete for yourself:
The application needs to be linked with the libsqlite3.so library.

It should run (provided you copy the generated db0 file from the “persist” project to the “view” project):
Names: tom, dick, harry, sue, abe, bobbie, gelsey, jeff, mike, psimon, winston

Name: Paul

Add Role: Mongo-man

Remove Role: DB guru
Task 2B: MySQL

The use of MySQL is the same as for CSCI110. Each Ubuntu computer in the laboratory runs its own instance of the MySQL database engine. Student accounts with the same name/password combinations exist on all machines. If you create tables on a particular machine, they are only present on that machine. Names and passwords for student accounts can be obtained in the laboratory.

There are different C++ drivers for accessing MySQL. The Oracle preferred version is MySQL Connector/C++. This is actually modelled on JDBC v4 and so its use will be familiar to students who have used JDBC in CSCI213 (of course, there are some differences such as the need to explicitly manage memory in your C++ code, and the fact that you aren't forced to put in all the try … catch constructs that Java insists upon – exceptions all bubble up to main() and terminate execution).

There are a few tutorials and references for MySQL Connector/C++ at the MySQL site. The example code below uses the same approaches as used in the sqlite version 2A. The connection to the database is managed by the MyRecordStore object; prepared statements are created, used, and destroyed rather than being recycled (which would be more efficient). The coding is very similar in style with explicit deletes of subordinate data tables, separate data fetches and so forth. The only real difference – the names of the classes and functions!

Project MyRecordMySQLPersist:

Create a new NetBeans project for the MySQL implementation. This should have a .sql file with the MySQL create table statements. Most of the code is, of course, copied from earlier tasks – classes: MyException, MyRecord (with the same addition of a removeRole() method as in task 2A), the similar main.cpp that creates a collection of MyRecord objects and loads them into the data store.

It's simply a matter of another variant of class MyRecordStore – this one using MySQL.

The tables should be defined in the local MySQL database system either by using the NetBeans generic database client with a connection that you have configured, or using the MySQL Workbench (obviously, you don't use my account!):
The MyRecordStore class will need sql::Driver and sql::Connection pointer data members:
There are a large number of separate header files for the MySQL Connector/C++ that must be included (some in the MyRecord.h file, others in MyRecord.cpp).

The connection is established in the constructor – the code is very similar in style to Java's use of JDBC with drivers, connections etc:
The code using the MySQL connection is a little more concise than that for SQLite (less code for failure handling – simply allowing any exceptions to terminate main()). The functions are all largely the same – prepare a statement, parameterise it, run it, use any returned data, delete it. (There are differences in the numbering schemes used to access elements in a returned row!)

```cpp
#include "MyRecordStore.h"
#include "cppconn/driver.h"
#include "cppconn/exception.h"
#include "cppconn/prepared_statement.h"

#include <cstring>
#include <stdlib.h>
#include <iostream>
using namespace std;

const char* databaseStr = "jdbc:mysql://127.0.0.1:3306";
const char* databaseUserName = "cabo";
const char* databaseUserPassword = "NotReallyMyPassword";
const char* databaseSchema = "cabo";

const char* notDB = "Sorry, but was unable to open database so operation failed";

const char* prepareFailed = "Failure when preparing statement";
const char* bindFailed = "Failure when binding arguments";
const char* writeFailed = "Failure when writing to database";

MyRecordStore::MyRecordStore() {
    // Constructors should always succeed. But suppose it failed and
    // the database couldn't be created?
    invalid = true;
    driver = get_driver_instance();
    try {
        dbcon = driver->connect(databaseStr, databaseUserName, databaseUserPassword);
    } catch (sql::SQLException &e) {
        // Failed
        return;
    }
    dbcon->setSchema(databaseSchema);
    // It's ok.
    invalid = false;
}

MyRecordStore::~MyRecordStore() {
    if (!invalid) {
        this->close();
    }
}

void MyRecordStore::close() {
    if (!invalid) {
        dbcon->close();
        delete dbcon;
    }
    invalid = true;
}

bool MyRecordStore::exists(const char* key) throw (const char*) {
    if (!invalid)
        throw (noDB);
    const char* countrecord = "select count(*) from MyRecord where _id=?";
    sql::PreparedStatement *psstmt = NULL;
    sql::ResultSet *rs = NULL;
    psstmt = dbcon->prepareStatement(countrecord);
    int count = 0;
    psstmt->setString(1, key);
    rs = psstmt->executeQuery();
    if (rs->next()) {
        count = rs->getInt(1);
    }
    delete rs;
    delete psstmt;
    return count == 1;
}
```

The exists() function is quite simple:
As is the allkeys() function:

```cpp
std::vector<std::string> *MyRecordStore::allKeys() {
    if (invalid)
        throw (noDB);
    std::vector<std::string> *vptr = new std::vector<std::string>();

    const char* selectall = "select _id from MyRecord";
    sql::PreparedStatement *psmt = NULL;
    sql::ResultSet *rs = NULL;

    psmt = dbcon->prepareStatement(selectall);
    rs = psmt->executeQuery();
    while (rs->next()) {
        string amid = rs->getString(1);
        vptr->push_back(amid);
    }
    return vptr;
}
```

The getInRole() function has a structure similar to that shown for the sqlite version with the obvious changes from sqlite3 function to "JDBC" style – the getInRole() function is left for you to complete.

The put() function is identical to that in the sqlite version – delete any existing version of the data, insert the new or updated version.

The deleteRecord() function is a little tidier than the sqlite version:
bool MyRecordStore::deleteRecord(const char* key) throw (const char*) {
    if (!valid)
        throw (newDB); // explicitly delete subordinate records in Phones, Addresses, others and Roles
    // then delete the myrecord entry
    // recreating the prepared statements for each operation - costly
    const char* deletephones = "delete from Phones where idPhone='" + id + ";";
    const char* deleteaddress = "delete from Addresses where idAddress='" + id + ";";
    const char* deleterelation = "delete from Relations where idRelation='" + id + ";";
    const char* deleteother = "delete from Other where idOther='" + id + ";";
    const char* deletemyrecord = "delete from MyRecord where idMyRecord='" + id + ";";

    // Pointer to unused part of sql string (?)
    sql::PreparedStatement *psmt = NULL;
    // Phones
    psmt = dbcon->prepareStatement(deletephones);
    psmt->setString(1, key);
    psmt->executeUpdate();
    delete psmt;

    // Addresses
    psmt = dbcon->prepareStatement(deleteaddress);
    psmt->setString(1, key);
    psmt->executeUpdate();
    delete psmt;

    // Other
    psmt = dbcon->prepareStatement(deleterelation);
    psmt->setString(1, key);
    psmt->executeUpdate();
    delete psmt;

    // Roles
    psmt = dbcon->prepareStatement(deleteother);
    psmt->setString(1, key);
    psmt->executeUpdate();
    delete psmt;

    // Finally...
    psmt = dbcon->prepareStatement(deletemyrecord);
    psmt->setString(1, key);
    psmt->executeUpdate();
    delete psmt;
    return true;
}

The `get()` method must access all the tables as in the sqlite version. The code to get data from the MyRecord and Role tables is shown here – you should be able to complete the function with the code needed to load phones, addresses, and other key/value data. (Note, MySQL is case sensitive – the createtables.sql script used MyRecord – so that is the table name, and myrecord won't work. There are also some differences in column names from those used in the sqlite version.)
MyRecord *MyRecordStore::get(const char* key) throw (const char*) {

    if (invalid)
        throw (noDB);
    // Again laborious
    // Load the MyRecord from its table, then add data from Other tables.
    const char* getrecord = "select * from MyRecord where _id=?";
    const char* getroles = "select role from Roles where idPerson=?";
    const char* getphones = "select type, number from Phones where idPerson=?";
    const char* getaddress = "select type, address from Addresses where idPerson=?";
    const char* getother = "select key, value from Other where idPerson=?";

    sql::PreparedStatement *psmt = NULL;
    sql::ResultSet *rs = NULL;

    psmt = dbcon->prepareStatement(getrecord);
    psmt->setString(1, key);
    rs = psmt->executeQuery();
    bool haveRecord = rs->next();
    if (!haveRecord) {
        delete rs;
        delete psmt;
        return NULL;
    }

    string id = rs->getString(1);
    string name = rs->getString(2);
    string email = rs->getString(3);
    string info = rs->getString(4);
    string image = rs->getString(5);

    MyRecord *arec = new MyRecord(id);
    arec->setName(name);
    arec->setEmail(email);
    arec->setInfo(info);
    arec->setImage(image);

    delete rs;
    delete psmt;
    // ------------------------------------------
    // Now ask about roles
    psmt = dbcon->prepareStatement(getroles);
    psmt->setString(1, key);
    rs = psmt->executeQuery();
    while (rs->next()) {
        string arole = rs->getString(1);
        arec->addRole(arole);
    }
    delete rs;
    delete psmt;
The `recordToTables()` function is again a sequence of steps writing appropriate data to each of the individual tables:

```cpp
void MyRecordStore::recordToTables(const MyRecord* data) {
    if (invalid)
        throw (noDB);

    // Insert the data first, then deal with Other data tables

    const char* putmyrecord = "insert into MyRecord values (?, ?, ?, ?, ?, ?)"
    const char* putrole = "insert into Roles values(?,?)"
    const char* putphone = "insert into Phones values (?, ?, ?)"
    const char* putaddress = "insert into Addresses values (?, ?, ?)"
    const char* putother = "insert into Other values (?, ?, ?)"

    sql::PreparedStatement *pstmt = NULL;
    pstmt = dbcon->prepareStatement(putmyrecord);
    pstmt->setString(1, data->getID());
    pstmt->setString(2, data->getName());
    pstmt->setString(3, data->getEmail());
    pstmt->setString(4, data->getInfo());
    pstmt->setString(5, data->getImage());
    pstmt->executeUpdate();
    delete pstmt;

    // Other
    if (data->getOtherKV().size() > 0) {
        pstmt = dbcon->prepareStatement(putother);
        map<string, string>::const_iterator it;
        for (it = data->getOtherKV().begin(); it != data->getOtherKV().end(); it++) {
            string akey = it->first;
            string avalue = it->second;
            pstmt->setString(1, data->getID());
            pstmt->setString(2, akey);
            pstmt->setString(3, avalue);
            pstmt->executeUpdate();
        }
        delete pstmt;
    }
}
```

Following sections of the code write out any role data, then phone data, then address data, and finally any other key/value data:

The project properties must add in the Qt include files and libraries as in earlier examples, and also add in the MySQL includes and link library code.
The program should run. The data added to MySQL can be viewed using MySQL Workbench (or the generic database client in NetBeans).

**Project MyRecordMySQLView:**

This project is almost identical to the MyRecordSQLiteView project – it's just it uses the MySQL version of class MyRecordStore. The GUI interface is as described for the sqlite version; the listmodel class is the same as that illustrated previously.

The project properties will need to specify the MySQL include lines and use of the MySQL library; the settings for these should be the same as for the MyRecordMySQLPersist project.

It should be easy to get to run:
Task 2 – completion (3 marks)

Demonstrate either your sqlite based data persistence implementation, or your MySQL based data persistence implementation.