An introduction to programming with libpqxx-object

Version 0.1.0

Roger Leigh
rleigh@debian.org

28th January 2004

Contents

List of Tables

1 Introduction

1.1 What is libpqxx-object

libpqxx-object is an extension to the libpqxx library, a C++ object-oriented interface to the PostgreSQL ORDBMS. While libpqxx provides objects for connecting to a database, transactions (executing queries) and result sets, the programmer still needs to use these objects directly in his code.

While developing a large application, I realised that I needed the database access to be rather more transparent than this: I wanted my classes to be able to interact with the database "behind the scenes", so that other programmers might use them without any need to know or understand SQL. In addition, I wanted to ensure *consistency*, so that slightly different (incorrect) queries would never be run at different places in the program following a change, and *control*, by restricting which database operations could be performed.

So, what does libpqxx-object actually do? In short, it encapsulates the database tables and rows as native C++ classes, so a programmer may do complex database work without even needing to know he is using a database.

1.2 Legal bit

2 libpqxx-object fundamentals

For each database table (or related set of tables), three classes are created:

1. A row class, representing a row in the table (or result set).
2. A table class, representing an entire table.
3. A row conversion class, used to convert a result set into a row class and a row into an SQL query.

The *row* class holds the information about all the columns in a single row, and has methods to get the data for each column and set the data for each column. This can be used to restrict access, for example to prevent modification of primary keys.

The *table* class "contains" the rows. It has methods for performing various database operations. These include various `SELECT` queries, defined as class methods, used to retrieve all or part of the table, or even individual rows. Rather than returning a result set, they return single or multiple row objects. In addition, methods are provided for all tables which wrap `INSERT`, `UPDATE`, and `DELETE` queries.

Using these two classes, it is possible to create or find row objects, manipulate them as needed and then insert into, update, or delete them from the database table.

The last class, the *row conversion* class, provides the "glue" needed to make this all happen: it knows how to construct a row class from an SQL result set, used by select queries. In addition, given a row object, it knows how to insert it into the database table, update it, or delete it from the database table.

Name	*Type*	*Constraints*	*Description*
id	serial	`PRIMARY KEY`	Primary key
name	text	`UNIQUE NOT NULL`	Place name
gridref	text	`NOT NULL`	OS grid reference

Table 1: `places` table structure.

That's all there is to it. libpqxx-object doesn't do anything you couldn't do yourself with libpqxx. What it provides are some classes to make this easier for you and, perhaps more importantly, gives all your classes a consistent interface.

3 Using libpqxx-object

It's not immediately obvious how to use the classes and template classes libpqxx-object provides. This is because most of them are not intended to be used directly: you derive your own classes from them.

To illustrate how it all fits together, this directory contains an example database and a program which uses the database using libpqxx-object.

3.1 Setting up the database

The example database structure is contained in the file `places.sql`, together with some sample data. Install the database using the following command (it requires a PostgreSQL installation with the postmaster running and `CREATE DATABASE` privilege):

```
$ psql -d template1 -f places.sql
```

The database created (called `libpqxx_object_tutorial`) contains a table called `places`. It's structure is shown in Table 1.

Our row class, representing a single row in this table, allows access to all these fields, with the exception of altering the id number, which is set by the backend database. If modification were allowed, unnecessary transaction failure could result.

The `UNIQUE` constraint on the name field isn't strictly necessary (there can be more than one place with the same name in reality), but is useful for the purposes of this tutorial.

3.2 Source code

The source code for the examples in the following sections is in the files `places.cc`, `places.h` and `places-main.cc`. Once you have run the `configure` script in the top-level directory, type `make` to build everything.

The `places-doc` subdirectory contains the `Places` API reference, generated from comments in the source[1]. Also take note of the libpqxx-object API reference in the top-level `doc/pqxx-object` directory.

[1] If you aren't using it already, consider this an example of why you should be using a tool like *Doxygen* to document your code.

The following sections explain what each part of the source code does, and why.

3.3 `Place` class

The `Place` class is the *row* class. It represents a single row of the `places` table. This is a normal C++ class, and does not use any libpqxx-object classes. It merely stores the column data for a single row.

Methods are provided to get the column data for all three columns. Methods are provided to set the data for the `name` and `gridref` columns. The `id` column data cannot be changed; this is allocated by the database when a row is inserted, and is effectively read-only.

Since this is a very simple class, its methods and data members should be self-explanatory.

The only complexity is here:

```
class Place
{
private:
  friend class PlaceConvert;
}; // class Place
```

The class `PlaceConvert` is a `friend` class. This is reqired so that the read-only *m_id* member can be set when constructing a `Place` from a result set (see next section).

3.4 `PlaceConvert` class

We need a way to convert the result of a `SELECT` query (a result set, of type `pqxx::result`) into a `Place` object. We also need a way to insert, update or delete rows in the `places` table using `Place` objects to describe the needed changes. The means to to this is the `PlaceConvert` class, which is the *row conversion* class. It is declared as follows:

```
class PlaceConvert : public pqxxobject::RowConvert<Place>
{
public:
  typedef pqxxobject::RowConvert<Place> row_base;
  PlaceConvert(pqxx::connection *connection,
               pqxx::transaction<> *transaction = NULL);
  Place *operator () (pqxx::result::const_iterator row);
  void operator () (Place& place,
                    row_base::operation_type operation);

}; // class PlaceConvert
```

`PlaceConvert` is derived from `pqxxobject::RowConvert<>`, a class template. Note that the template parameter is the *row* type we are using for our conversions; in this case, it is the `Place` class.

The constructor requires a pointer to a `pqxx::connection` database connection object, and optionally a `pqxx::transaction<>` transaction object (required by the `pqxxobject::RowConvert<>` constructor, since we are going to be doing database work).

Two operator () methods are declared (they are declared as *pure virtual* functions in the base class). That is, this class is a *function object* that can behave like a function, as well as a class. The former takes an argument of a `pqxx::result::const_iterator` (a pointer to a row in a result set) and returns a pointer to a `Place` object, allocated with `new`. This is called whenever a `Place` needs to be created. The latter takes a `Place` and an `operation_type` as arguments. The `operation_type` is one of `row_base::OPERATION_INSERT`, `row_base::OPERATION_UPDATE`. or `row_base::OPERATION_DELETE`, and tells the function whether the row should be inserted, updated, or deleted (`row_base` is a `typedef` for our convenience).

Take a look at `places.cc` to see the definition of these functions. The former function is very simple: we just create a new `Place`, and then assign the result set values to the data members.

The latter function is more complex. Depending on which `operation_type` was selected, a different query is constructed using a `std::ostringstream`. Finally, the query is run with `exec_query_autocommit()` (this is provided by `pqxxobject::Transaction`, which `pqxxobject::RowConvert<>` derives from).

So, we now have the means to create `Place` objects from the database `places` table, and the means to insert, update, and delete rows.

3.5 `PlaceTable` class

The `places` database table is represented by the `PlaceTable` class. This provides all the operations you can perform on a table: selecting rows, and inserting, updating, and deleting rows.

```
class PlaceTable : public pqxxobject::Table<Place, PlaceConvert>
{
public:
  typedef pqxxobject::Table<Place, PlaceConvert> table_base;
  PlaceTable(pqxx::connection *connection,
             pqxx::transaction<> *transaction = NULL);
  virtual ~PlaceTable();
  enum sort_order
    {
      ORDER_ID,
      ORDER_NAME,
      ORDER_GRIDREF
    };
  std::vector<Place> *get_list(sort_order order = ORDER_NAME);
  Place *find(int place_id);
  Place *find_name(const std::string& name);
  std::vector<Place> *find_gridref(const std::string& gridref);
}; // class PlaceTable
```

`PlaceTable` is derived from `pqxxobject::Table<>`, a class template. The template parameters are the *row* type and the *row conversion* types; in this case these are `Place` and `PlaceConvert`, repectively.

The constructor requires a pointer to a `pqxx::connection` database connection object, and optionally a `pqxx::transaction<>` transaction object (required by the `pqxxobject::Table<>` constructor, since we are going to be doing database work).

Methods are provided to perform various SELECT queries:

- get_list() gets all rows, sorted according to the sort_order specified.
- find() gets the row with the corresponding id number.
- find_name() gets the row with the corresponding name.
- find_gridref() gets the rows with the corresponding gridref.

Functions that return zero or one rows return a pointer to a Place allocated with new, or NULL if no row was found. Functions that return zero or many rows return a pointer to a vector of rows.

```
namespace pqxxobject
{
  template<typename _Row, typename _Convert>
  class Table : public Transaction
  {
  public:
    typedef _Row row_type;
    typedef _Convert convert_type;
  protected:
    Table(pqxx::connection *connection,
          pqxx::transaction<> *transaction = NULL);
  public:
    virtual ~Table();
    virtual void insert(row_type& row)
      { convert(row, convert_type::OPERATION_INSERT); }
    virtual void update(row_type& row)
      { convert(row, convert_type::OPERATION_UPDATE); }
    virtual void erase(row_type& row)
      { convert(row, convert_type::OPERATION_DELETE); }
    virtual
    row_type *find_one(const std::string& query)
    virtual
    std::vector<row_type> *find_many(const std::string& query)
    convert_type convert;
  }; // class Table
}; // namespace pqxxobject
```

The base class pqxxobject::Table<> provides generic insert(), update() and erase() functions. (erase() is not called delete for hopefully obvious reasons.) Notice how they are simple wrappers around our PlaceConvert class. The functions find_one() and find_many() are used by our SELECT methods. Since they are templated, they know which row type to return, and automatically call our RowConvert type to create them.

3.6 Putting it all together

The file places-main.cc is a driver program to show our Places class in action. It opens a database connection, creates a PlaceTable object and then runs various SELECT queries followed by an INSERT, UPDATE, and DELETE. The code is commented, to show what is happening.

Run the program like so:

```
$ ./places
```

You should see the results of all the actions displayed (page with `more` or `less` to view it all). Any errors should cause an exception to be thrown, which will be caught and displayed before the program exits.

4 Going further

The wrapping of simple rows and tables is (I hope) fairly simple, if a little long-winded. This should be easy to speed up if you create some template files with skeleton header and source (perhaps derived from `places.h` and `places.cc`).

There are more complex situations, such as where is doesn't make sense for a row of a table to exist as an object in its own right. For example, if we have a `accounts` table, holding account information, and then have `customer_accounts` and `staff_accounts` which reference it, we could say that a `customer_acount` *is an* `account` (i.e. the relationship could be expressed through inheritance).

In this case, we could create an `Account` *row* class, along with the expected `AccountConvert` *row conversion* class and an `AccountTable` *table class*. When we come to the `CustomerAccount` class, this can derive from the `Account` class like so:

```
class CustomerAccount : public Account
{
  CustomerAccount(const Account& account);
}; // class CustomerAccount
```

How would we implement the `CustomerAccountConvert` class? Simple: we do the conversion from a result set to `CustomerAccount` in two stages: Firstly, we create an `Account` object using its `AccountConvert` class (this requires the column names of the result set to be what it expects, which could have implications on your `SELECT` query design). Next, we pass this to our new `CustomerAccount` object as a constructor argument. Finally, we assign result set values to the member data of the `CustomerAccount` object as before.

Now, how about handling database changes? This will vary depending on the situation, but you could get an `AccountConvert` object and do the database operation on the `Account` first, then repeat for the `CustomerAccount` part. Alternatively, you might wish to avoid any updates to the base class.

The beauty of using inheritance in this way is that the database relations are expressed while using the *row* objects. You can pass a `CustomerAccount` to any method requiring an `Account`, so having the row represented as a C++ object does pay off.

Have Fun!
Roger Leigh

5 Further Reading

The libpqxx library includes a *Doxygen*-generated API reference, and the code in the test-suite (in the `test` subdirectory of the libpqxx source) provides

useful examples of how to use libpqxx. You'll need to understand how to use the `pqxx::connection`, `pqxx::transaction<>` and `pqxx::result` objects in order to use libpqxx-object.

The PostgreSQL database documentation will come in handy if you're administering PostgreSQL. There's also an SQL command reference and the libpq API reference is also useful (since libpqxx wraps it).