An introduction to programming with libpqxxobject

Version 0.1.0

Roger Leigh
rleigh@debian.org

22nd May 2004

Contents

List of Tables

1 Introduction

1.1 What is libpqxxobject

libpqxxobject 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 libpqxxobject 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 libpqxxobject fundamentals

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

1. A row class, representing a row in the table (or result set).
2. A table class, representing an entire table.

The *row* template 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. In addition, virtual methods are provided to insert, update, erase and refresh rows (these correspond to `INSERT`, `UPDATE`, `DELETE` and `SELECT` queries), and should be overridden where required (the default is to do nothing).

The *table* template 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 table class is optional–it exists to provide a consistent interface for running queries, but isn't strictly required to insert, update and delete rows.

In addition, a *field* template class is provided to use within a row class. Its purpose is to contain a field value (a column value in a row).

That's all there is to it. libpqxxobject 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.

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.

3 Using libpqxxobject

It's not immediately obvious how to use the classes and template classes libpqxxobject 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 libpqxxobject.

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 libpqxxobject API reference in the top-level `doc/pqxxobject` directory.

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

[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.

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, which is derived from `pqxxobject::row` (and hence also `pqxxobject::row_base`). It merely stores the column data for a single row, and also provides methods to insert, update, delete and refresh the row data (using libpqxx).

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.

3.4 `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>
{
public:
  PlaceTable(pqxxobject::transaction& tran);
  virtual ~PlaceTable();
  enum sort_order
    {
      ORDER_ID,
      ORDER_NAME,
      ORDER_GRIDREF
    };
  row_list_ptr get_list(sort_order order = ORDER_NAME);
  row_ptr find(int place_id);
  row_ptr find_name(const std::string& name);
  row_list_ptr find_gridref(const std::string& gridref);
}; // class PlaceTable
```

`PlaceTable` is derived from `pqxxobject::Table`, a class template. The template parameter is the *row*, in this case is `Place`.

The constructor requires a reference to a `pqxxobject::transaction` transaction object since we are going to be doing database work). This object encapsulates a `pqxx::connection` and `pqxxobject::transaction`, and provides facilities for recursive object serialisation.

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`.

These return either a `row_ptr` or `row_list_ptr`. These are classes based on `std::auto_ptr` which are defined by the `pqxxobject::table` base class. This allows for safe allocation and deletion of row objects, providing the usual caution is kept when using `std::auto_ptr`

```
namespace pqxxobject
{
  template<typename _Row,>
  class table : public Transaction
  {
  public:
    /// The type of the row the table contains.
    typedef Row row_type;
    typedef typename Row::row_ptr row_ptr;
    typedef std::list<Row> row_list;
    typedef std::auto_ptr<row_list> row_list_ptr;
  protected:
    table(pqxxobject::transaction& tran);
  public:
    virtual ~table();
    virtual void insert(row_type& row)
      { row.insert(m_transaction); }
    virtual void update(row_type& row)
      { row.update(m_transaction); }
    virtual void erase(row_type& row)
      { row.erase(m_transaction); }
    virtual void refresh(row_type& row)
      { row.refresh(m_transaction); }
    virtual row_ptr find_one(const std::string& query);
    virtual row_list_ptr find_many(const std::string& query)
    pqxxobject::transaction& m_transaction;
  }; // 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 the `row` 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.

3.5 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 it 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 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 `convert_impl()` function? Simple: we initialise the base class by calling its `convert_impl()` method, from within our `convert_impl()` method. This could have implications on your `SELECT` query design). I would recommend creating views to simplify your code: this can handle making the column names unique and can also automate the joins.

Now, how about handling database changes? This is also straightforward: in your `insert_impl()`, `update_impl()` and `erase_impl()` row methods, delegate the work to the appropriate row methods in the base class (or contained object if using containment rather than inheritance).

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 libpqxxobject. libpqxx also includes a very good tutorial, which I highly recommend.

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).