op_lite

a portable, lightweight object persistence library for C++

White paper draft – December 2014

Carsten Arnholm

op_lite license:


```
/*
** Author: Carsten Arnholm, November 2014
** This code was written for my weather station project [2010-> ]
**
** This code follows the sqlite3 license model, i.e.:
** The author disclaims copyright to this source code.  In place of
** a legal notice, here is a blessing:
**
**    May you do good and not evil.
**    May you find forgiveness for yourself and forgive others.
**    May you share freely, never taking more than you give.
*/
```

Table of Contents

Introduction....3
What is op_lite?....4
Front end....4
Back end....5
Architecture overview....6
Writing an application using op_lite....7
Application philosophy....7
Declaring a user defined persistent class....8
Declaring persistent member variables....9
Initialisation of persistent members....10
Mapping persistent members in database....11
Persistent object identifiers....11
Persistent classes and the type factory....12
Creating a new database....13
Opening an existing database....13
Database transactions....14
Nested transactions....14
Instantiating persistent objects....15
Deleting persistent objects....15
The transient object cache....16
Database root objects....17
Creating a new root object....17
Restoring an existing database root object....18
Polymorphic roots....18
Persistent polymorphic pointers....19
Ownership of objects....19
Persistent containers....20
Persistent containers and polymorphic pointers....20
Examples....21
Example "op_demo" – a persistent triangle....21
Point class, header....22
Point class, implementation....23
Line class, header....24
Line class, implementation....25
Triangle class, header....26
Triangle class, implementation....27
op_demo class, header....28
op_demo class, implementation....29
Main program....32
Building op_lite....35
op_lite dependencies....35
Building with 'Makefile.msvc' on Windows....35
Building with 'Makefile' on Linux....36
References....37

Introduction

In the world of C++, a large number of excellent open source libraries exist for almost any conceivable purpose. In addition, most of these libraries are cross platform, i.e. they may be used under several different operating systems. These libraries often exploit the power of object orientation as defined in C++.

The situation is not quite the same for database libraries. In this author's opinion, it is a problem that the word 'database' for many people is synonym with a traditional relational database based on some form of SQL. Although these databases are powerful, the programming model they impose is not supporting object oriented programming.

Such databases tend to be oriented towards a client/server architecture, suitable for applications dominated by a relatively large number of simple and independent transactions, e.g. consider bank withdrawals/deposits as representative for this type of database. This is not a suitable model in all cases, especially not for applications relying on more complex data structures and transactions, possibly with large amounts of data.

What is missing is an open source and portable database library supporting object orientation, allowing the developer to use native C++ classes with persistent instances living naturally in the database. Such systems do exist, most notably /1/, but there are unfortunately few open source libraries in this 'pure' category, if any.

op_lite's objective is therefore to provide a single process object oriented database approximating /1/, with support for persistent containers and polymorphic pointers. If there is enough interest, op_lite can also become become an open source library.

The term "approximate" means that some compromises will have to be applied since writing a complete OO database from scratch is beyond the scope and resources available to this author. However, the compromises appear to be acceptable to the author.

One may consider /3/ an alternative to op_lite, but in the author's opinion such alternatives are too heavy weight, complicated and does not succeed adequately in hiding the SQL relational database. It seems to adopt the traditional relational database viewpoint, whereas op_lite tries to limit the databases obligation into providing reasonably efficient and simple object persistence, and at the same time support central OO topics such as polymorphism.

The next section outlines the strategy chosen for op_lite and the compromises made for the library.

What is op_lite?

The name op_lite stands for *Object Persistence – Lightweight*. It is a C++ library that offers automatic in-process object persistence of C++ objects, the application code never explicitly reads or writes to the database – it all happens behind the scenes, given that certain programming patterns are followed. This is similar to most other "real" OO databases. The effect is that the objects are perceived to “live in the database”.

Front end

op_lite is implemented as a small C++ class library. The library provides helper classes for managing databases, base classes for deriving user defined persistent classes, and classes for describing persistent data.

The front end is generally the only layer an application writer needs to relate to. The front end contains the following few C++ classes

Class name	Description
op_manager	Object API for creating and opening/closing database(s). The op_manager is a singleton object accessible via the function op_mgr().
op_type_factory	An object managed by op_manager. The op_type_factory registers the persistent types, thereby allowing polymorphic persistent objects.
op_persistent_class<T>	Instantiate this class and install in op_type_factory for every persistent class in the application.
op_database	Object representing a single database file. The main purpose is to have an object to pass around, often the application does not use this object directly.
op_root<T>	Object representing a named root in the database of user defined type. Database roots are typically the first entry points when reading an existing database.
op_object	Base class that user defined classes derive from to implement persistent objects.
op_value<T>	Template for persistent member variables. There are typedefs for the most common variants, such as *op_int*, *op_double* and more.
op_vector<T>, op_map<T>	Persistent container classes similar to std::vector and std::map. Like op_value<T>, these containers can also be persistent member variables.
op_ptr<T>	Shared, persistent polymorphic pointer to object of type T. An op_ptr<T> can be dereferenced to obtain a T*, where T* may be an abstract base class.
op_unique_ptr<T>	Unique (=owned), persistent pointer to object of type T
op_new<T>	Class acting as "persistent new" for object of type T. There is also op_delete(op_object* obj) for deleting persistent objects
op_transaction	Object for managing (optionally nested) database transactions

An application-writer using op_lite generally only needs to learn how to understand and use the above classes. Almost everything else is implementation detail. This will become clearer with some some source code examples later in this document.

Back end

The "back-end" of op_lite (meaning the low level management of the database) is handled by SQLite /2/. SQLite is a software library that in its own words "*implements a self-contained, serverless, zero-configuration, transactional SQL database engine*". SQLite is the most widely deployed SQL database engine in the world.

Given the introduction in this document, some may consider the use of SQLite a strange choice and a huge compromise, not much different from other libraries. Admittedly, using SQLite as back end is indeed a compromise, but as it will be shown, the programming model using op_lite is very different and much simpler than typical alternative libraries wrapping SQLite.

As mentioned in the introduction, the goal in op_lite is to hide the underlying SQLite back end as much as possible and offer a natural OO interface for persistent objects instead.

Some positive aspects of SQLite, despite of its SQL model, is that it is self-contained and serverless. It has a C/C++ API that is efficient and the size of the databases it can handle is virtually unlimited. This latest point is very important, as for example /1/ may be limited by the allowable virtual memory that the application is able to address.

Therefore, by encapsulating SQLite completely and providing automatic ways of mapping a user defined C++ object into the database, an efficient and approximate "OO-like" database can nevertheless be realised, as will be shown.

There are also additional benefits in using SQLite as back end. One such benefit is that it is possible to browse the database using standard SQLite tools, for example when debugging, plus it is possible to perform special database operations, e.g. merge 2 databases, using standard SQLite tools. This is not always relevant, but in some cases it can be very useful.

All in all, the reasons for using SQLite as back end are

- it is a zero-configuration, in-process engine
- it is proven technology, extensively tested
- it is very efficient
- it is open source
- it is very well documented
- it is portable (both source code and databases)
- it supports virtually unlimited size databases
- it allows using standard SQLite tools for special purpose operations
- it relieves the author of op_lite of developing a competing back end

I.e. given limited resources for development, this compromise is acceptable as we are able to build

on many years of development work in SQLite. The secret is to find a way to use it efficiently and transparently as a back end for an OO database.

Architecture overview

The library architecture is fairly simple, but the simplest possible description is that a C++ application implements one or more persistent classes that 'live in the database', ref. class1-3 in figure below.

These classes are implemented according to the standard op_lite implementation pattern and are thus automatically reflected in the database, eliminating the need for explicit read and write operations.

The op_lite library itself consists of 3 conceptual layers of code

- Top : the op_lite layer that is visible to and used by the application developer
- Middle: "sqLayer", SQL-oriented layer for mapping of user objects to database.
- Bottom: sqLite layer. A standard sqlite3 source file compiled into the library.

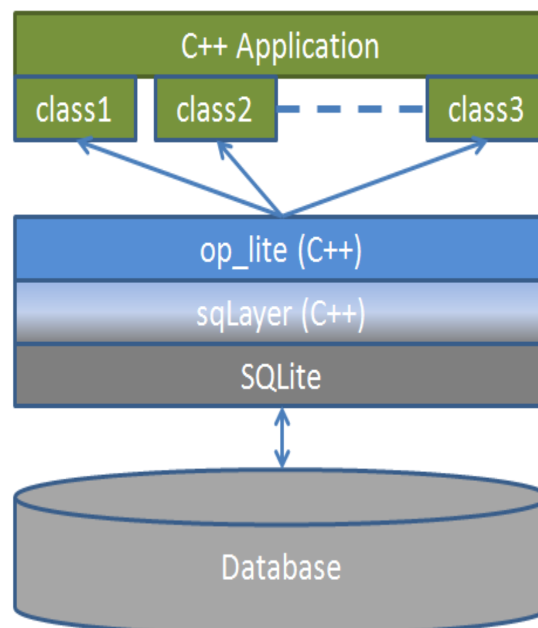


Illustration 1: op_lite architecture sketch

Implementing a user defined persistent class means to derive the user defined class from an op_lite base class (*op_object*) and to declare its member variables in a special manner, so these values can be transparently captured to/restored from the database.

Each user defined application class has a unique name and will automatically be mapped to a database table with the same name. Each persistent member variable of the user defined class will be mapped to a column in the same database table, where the column names are the same as the member variable names. These members include standard value types, but also pointers to other objects in the database, and even container classes containing data or pointers to other persistent

objects.

The op_lite API has functions for creating and opening databases. There is support for nested transactions ensuring both efficient i/o and a guarantee that object modifications will be reflected in the database.

An important difference between the op_lite model and serialization techniques is that in op_lite, i/o will only happen for the objects explicitly being referred to, whereas in serialization all objects will be subject to read/write. This can have important performance implications.

Writing an application using op_lite

This chapter explains the steps required for writing an application with persistent C++ classes based on op_lite, using short code snippets along with some discussion. The code in this section is not complete examples, but instead focusing on the specific rules to be followed.

Writing an application with user defined persistent classes according to the pattern required by op_lite is not very different from writing ordinary transient C++ classes, but there are some special requirements and it is useful to summarize the main requirements before studying the details:

- C++ RTTI must be enabled for all components using op_lite
- C++ exception handling should be enabled to handle error situations that may occur
- Persistent classes must directly or indirectly be derived from op_object
- Persistent classes must provide default constructors
- Persistent classes must implement the op_layout(...) virtual member function
- Persistent classes must use special member variables designed for persistence
- Persistent member variables must be initialised using op_lite macros
- Persistent objects must be instantiated using op_new<T>, where T is the persistent type.
- Persistent classes must be registered in the type factory to allow restoration via pointer

This may seem like much, but in practice it is not very different from standard programming rules, and with some training it works rather well in practice.

Application philosophy

In the standard OODB paradigm, the persistent classes are the same as the application classes, there is no difference. The examples in this document reflect this view, but it is not the only possible way to use op_lite. One may for example have “pairs of transient and persistent classes” and use the persistent classes strictly for database communication. In the author's view, such patterns defeats important aspects of the OODB paradigm, so it is not further discussed here.

Therefore, when reading the following chapters, the classes presented should be understood as the actual application classes, i.e. the application objects should be thought of as residing in the database directly.

Declaring a user defined persistent class

When creating a persistent C++ class using op_lite, the user defined class must inherit from op_object, either directly or via other abstract or concrete persistent classes. For example, assuming you want to create a persistent class Point, you need to do the following

```
#include "op_lite/op_object.h"


// Persistent Point
class Point : public op_object {
public:
   // op_lite: default constructor is required
   Point();

   // construct point from coordinates
   Point(double x, double y);

   // destructor
   virtual ~Point();

   // this is a required overload on all op_lite user objects
   void op_layout(op_values& pvalues);

   // ... other member functions
};
```

From the Point class, we observe 3 general rules that apply in the class header:

A user defined persistent class must ...

1. ... inherit from op_object, or from another class derived from op_object.
2. ... have a default constructor. It may have other constructors.
3. ... override the pure virtual function op_layout. More about this later.

Declaring persistent member variables

In the previous section we learned the basics of deriving a persistent class. However, the class will normally also have persistent data members, for example the coordinates of a point:

```
class Point : public op_object {
public:
   // ... class member functions
private:
   op_double m_x;   // persistent double: x-coordinate
   op_double m_y;   // persistent double: y-coordinate
};
```

The above illustrates how to declare persistent member variables in a persistent class, in this case 2 double values, m_x and m_y.

The following persistent member variable types are supported

- op_int : integer
- op_double : double
- op_string : std::string
- op_blob : BLOB (Binary Large Object)
- op_ptr<T> : shared pointer to object of type T
- op_unique_ptr<T> : unique (=owned) pointer to object of type T
- op_vector<T> : vector template container, where T is any built in transient C++ type or even op_ptr<T>
- op_map<K,V> : map template container, with features similar to op_vector<T>

These op_* variable types are specialisations of the template op_value<T>, except for the template containers. If required, new variable types may be declared. The use of blobs and persistent pointers will be described later.

Observe the importance of for example being able to declare and use a persistent member variable of type **op_vector<op_ptr<Shape>>,** where Shape is some abstract user defined persistent base class. This means that a persistent object can have a member variable that is a container of polymorphic pointers to other objects in the database. More about this later.

A persistent class may also have ordinary transient member variables, but please observe that such variables will NOT be stored in the database, and the values will not be re-initialised when an object is retrieved from the database. In most cases, all member variables are persistent in a persistent class.

Initialisation of persistent members

In the previous section we learned how to declare persistent member variables in the class header file. However, to complete the process, the members need to be initialised in a special way in each constructor, using the macros `op_construct` used in the default constructor where no special value initialisation is needed (default constructors of the members assumed) or `op_construct_v1`, where a single value may be used to initialise the member.

```
Point::Point()
: op_construct(m_x)
, op_construct(m_y)
{}

Point::Point(double x, double y)
: op_construct_v1(m_x,x)
, op_construct_v1(m_y,y)
{}
```

These macros ensure that persistent member variables are initialised with the proper value, and they ensure that the variable name is recorded for use in the database (as table column names for those who are interested in the details).

op_construct initialises the member using its default constructor. op_constructor_v1 initialises the member using the given single variable.

The result is that the Point class will be represented in the database as a table with name Point, having 2 double value columns named m_x and m_y respectively.

Observe that there is also a couple of variant forms, namely `op_construct_pk` and `op_construct_pk_v1` which are used when the member variable is to be explicitly declared as "primary key" in the database. This is an optimisation scheme, example of its use can be found in the standard op_root<T> class template.

Mapping persistent members in database

In the previous section we learned how to declare and initialise persistent member variables in a user defined persistent class. That information was sufficient to declare the database schema, but not sufficient to instantiate or restore individual objects in the database.

To enable the objects of this class to be transparently saved and restored to/from the database, one more step is required: We have to "bind" the member variables so their values can be saved and restored. This is done using the `op_bind` macro, within the virtual member function `op_layout` that all persistent classes must implement:

```
void Point::op_layout(op_values& pvalues)
{
   op_bind(pvalues,m_x);
   op_bind(pvalues,m_y);
}
```

As can be seen from the above, one simply mentions each persistent member variable in the class. This function will be called by op_lite and relieves the application writer of having to save/restore data explicitely. This is a huge benefit of the OO paradigm.

With this declaration, the Point class is complete (except for its special application-functionality, unrelated to op_lite) and as written it is automatically saved/restored in the database when needed.

Persistent object identifiers

A standard, transient C++ variable on the heap is identified using its transient memory address

```
int* hundred = new int(100);
```

Here, `hundred` is a pointer to an integer value. The pointer value is the memory address where the value 100 is found.

Similarly, we may have a persistent Point object 'pnt' which also has a transient memory address:

```
Point* pnt = op_new<Point>(-1.0, 0.0);
```

The transient memory address of the Point object 'pnt' will <u>not</u> generally be the same in different sessions, it must be assumed to take a new transient address each time. This is important to understand.

However, the *persistent object identifier* will be the same in different sessions and can be used to identify the persistent object, even when the transient pointer address has changed.

```
op_pid id = pnt->pid();
```

Observe that the pid is unique only within its own class, two objects of different concrete classes may share a common pid value. To uniqely identify the object, both the class name and pid value are required.

Persistent classes and the type factory

In the previous sections, we have illustrated the basics of declaring a persistent C++ class, i.e. how to write constructors, member variables etc. It has also been mentioned that op_lite supports polymorphic pointers, *op_ptr<T>,* either as direct members of another persistent class, or as member of a persistent container, such as *vector<T>* or *map<K,V>*. This is very convenient and will be covered in more detail later.

When e.g. a *op_ptr<Point>* is stored in the database, the pointer is represented as a text:

```
0 Point 123
```

The first value (zero) indicates the format, the second value (class name) indicates the concrete type of the object, and the third value is the object's persistent identifier, or in more concrete terms its `rowid` in the SQLite database "Point" table.

When restoring objects from a database by dereferencing a persistent pointer such as *op_ptr<Point>*, one cannot be sure that the object is in fact a point by looking at the transient variable. Maybe the Point derives from a Shape and our pointer variable is really a *op_ptr<Shape>*. If this is the case, we should still get a Point returned when the *op_ptr<Shape>* is dereferenced.

For this to work, we need the help of the *type factory*, which solves the problem of turning the string "Point" onto an object of type *Point*. This is achieved by installing all persistent types in the type factory at the beginning of each program session, before opening any database. The process is a rather simple one line of code per class, for example:

```
#include "op_lite/op_manager.h"

#include "Point.h"
#include "Circle.h"
#include "Line.h"
#include "ShapeCollection.h"


void poly_shapes::init()
{
   // register the persistent types in the type factory
   op_mgr()->type_factory()->install(new op_persistent_class<Point>());
   op_mgr()->type_factory()->install(new op_persistent_class<Circle>());
   op_mgr()->type_factory()->install(new op_persistent_class<Line>());
   op_mgr()->type_factory()->install(new op_persistent_class<ShapeCollection>());

}
```

The example shows how to install 4 user defined persistent classes (*Point, Circle, Line, ShapeCollection*) as persistent classes in the type factory. This code must mention all non-abstract persistent classes in the application, and the code should be executed in every session, before reading or writing a database. Luckily, this is all an application developer needs to do regarding the type factory.

Observe that you can easily extend the number of persistent classes in an existing database, but once a persistent type has been added, it should not be removed. If a persistent class is missing in the type factory, an exception will be thrown when such an unknown object is encountered.

Creating a new database

In most applications, a single database is used, although it is possible to create and use multiple databases at the same time. Each database is identified with a logical name (a string), and is managed by the op_manager singleton class, conveniently accessed via the function `op_mgr()`

We may thus create a database with the logical name "demo":

```
string  db_path = "~/demo.db";
if(op_database* db = op_mgr()->create_database("demo",db_path)) {

   cout << "OK, created database " << db_path << endl;
   // ... use this database

}
```

The above shows how we may create a new database via `op_mgr()` and receive a pointer to an `op_database` instance. If the pointer is non-null, the operation was successful and we may use the database by storing objects in it.

Creating a new database makes that database the currently selected database.

Opening an existing database

Opening an existing database is very similar to creating a new one:

```
string  db_path = "~/demo.db";
if(op_database* db = op_mgr()->open_database("demo",db_path)) {

   cout << "OK,  opened existing database " << db_path << endl;
   // ... use this database

}
```

Opening an existing database makes that database the currently selected database.

There is also a similar `open_create_database` which does the obvious thing.

Database transactions

Database transactions are used for grouping together database operations and thereby achieving optimisation and consistency. It is also an important tool to simplify the logic of creating and retrieving data from op_lite databases.

Transactions are managed by `op_transaction` objects which exist on the application stack (i.e. they are **not** instantiated with *new* or *op_new<T>*). Once the transaction object goes out of scope, the affected objects in that scope are saved or updated in the database and their transient representations are deleted:

```
string  db_path = "~/demo.db";
if(op_database* db = op_mgr()->create_database("demo",db_path)) {
   cout << "OK, created database " << db_path << endl;

   // use a transaction to manage the objects created in this scope
   op_transaction trans(db);

   // create persistent objects here
   Point* pnt1  = op_new<Point>(-1.0, 0.0);

   // when the trans object goes out of scope, the pnt1 memory object
   // will no longer be valid, but the point will remain in the database
}
```

It is efficient to group together creation/modification of several objects within the same database transaction. It also helps to manage the transient cache of objects. Every time a persistent object is created in the database or restored from it, a transient cached object is also created. If such cached objects were not created in the context of a transaction, they would have to be manually deleted. In most cases, it is safer and easier to let the transaction object manage the cache, as in the example above.

Nested transactions

Database transactions may be nested, to manage the cached objects and database transactions in a more advanced manner:

```
if(condition1) {
  op_transaction trans1(db);
  Point* pnt1  = op_new<Point>(-1.0, 0.0);

  while(condition2) {
     op_transaction trans2(db);
     Point* pnt2  = op_new<Point>(1.0, 1.0);
  }
}
```

Instantiating persistent objects

User defined persistent classes may be instantiated (created) in the currently selected database. As it was illustrated in the previous section, we instantiate new persistent objects in the currently selected database using the template op_new<T>. The syntax for instantiating persistent objects becomes

```
// create a persistent point at origin (x,y) = (0,0)
Point* origin  = op_new<Point>(0.0, 0.0);

// we may now call member functions on the object
origin->print(0);

// delete the transient memory object, but keep the object in the database
delete origin;
```

Up to 10 constructor parameters are supported by op_new.

It is recommended to always encapsulate such object instantiation in a database transaction, as shown in the previous section. Then explicit deletion of transient memory object is not required as it will be handled by the transaction, and at the same time any changes to objects during the transaction will be reflected in the database, unless the transaction is a read-only transaction.

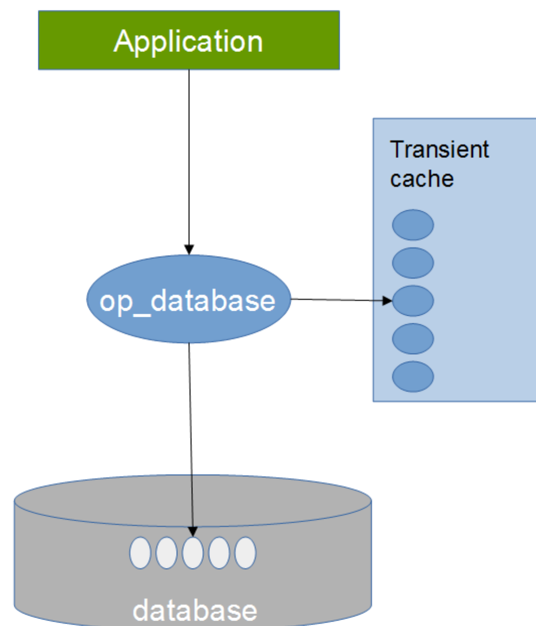
Deleting persistent objects

If the application wants to delete a persistent object from a database, it may be done by first obtaining the persistent object:

```
// create a persistent point at origin (x,y) = (0,0)
Point* origin  = op_new<Point>(0.0, 0.0);

// origin now exists in the database and in memory

// delete the object in database and transient cache
op_delete(origin);

// persistent object "origin" no longer exists in database or memory
```

The transient object cache

When an object is created in the database via its `op_database`, or restored from it, the object is also represented as a transient object owned by the `op_database` cache.



During an `op_transaction`, the cache will be filled with objects as new persistent objects are created, or existing ones are restored from the database.

When the `op_transaction` destructor executes, the persistent objects are automatically updated and the corresponding cached objects are then deleted. If the cache was empty before the transaction, the cache is again empty after the transaction.

If a cached object is deleted (using C++ `delete`) during a transaction, the deleted object is also removed from the cache.

Observe that there is a difference between deleting an object implicitly as part of a transaction and doing it explicitly using C++ `delete`. When using C++ `delete`, the object will be removed from the cache, discarding possible changes to the object.

To delete an object explicitly and achieving the same effect as when using transactions, an explicit update_persistent() must be performed prior to deleting.

Example without transaction:

```
    Point* point = ...
    point->set_x(2.5);
    point->update_persistent();   // update in database
    delete point;                 // delete transient object
```

Example with transaction:

```
{
    op_transaction trans(op_mgr()->selected_database());
    Point* point = ...
    point->set_x(2.5);
}                                 // transaction updates & deletes
```

It is clearly faster, safer and more elegant to use transactions to manage correct object updates in the database, as well as manage the object cache.

Database root objects

Object oriented databases often rely on special objects called root objects that contain a pointer to another persistent user object. By restoring a root object, one is able get access to other objects in the database. This is a slightly different pattern than query-based data access common in relational databases. In op_lite, both methods are available.

A database may have one or several root objects, but the number is typically small. Each root has a unique name string that is used when restoring it in a later session. The root object then typically points to some user defined persistent object containing a container of polymorphic object pointers. This way, one may navigate through the objects in the database, using a root as starting point.

Database roots objects are instantiations of the template `op_root<T>`, where T is a user defined persistent class.

Creating a new root object

Let us assume we have a database with geometrical objects, points, lines and triangles. A triangle contains lines describing its edges, and lines contain points to describe its ends. We may for example use a Triangle object as a database root and thus get access to all information in the triangle:

```
if(op_database* db = op_mgr()->create_database("demo",db_path)) {

   // use a transaction to clean things up + it is more efficient
   op_transaction trans(db);

   // create 3 persistent Points
   Point* pnt1  = op_new<Point>(-1.0, 0.0);
   Point* pnt2  = op_new<Point>(+1.0, 0.0);
   Point* pnt3  = op_new<Point>( 0.0,+1.0);


   // create 3 persistent Lines, referring to above points
   Line* l1     = op_new<Line>(pnt1,pnt2,"Line1");
   Line* l2     = op_new<Line>(pnt2,pnt3,"Line2");
   Line* l3     = op_new<Line>(pnt3,pnt1,"Line3");

   // One persistent Triangle, referring to above lines
   Triangle* t1 = op_new<Triangle>(l1,l2,l3);

   // establish the Triangle as a root in the database
   // and use "TRIANGLE_ROOT" as the root name
   op_root<Triangle>* root = op_new<op_root<Triangle>>("TRIANGLE_ROOT",t1 );

   cout << endl << "Printing new triangle" << endl;
   t1->print(0);

}

// close database file
op_mgr()->close_database("demo");
```

Restoring an existing database root object

Restoring a database root is straightforward

```
if(op_database* db = op_mgr()->open_database("demo",db_path)) {
   // use a transaction to clean things up + it is more efficient
   op_transaction trans(db);
   if(Triangle* t = op_root<Triangle>::restore("TRIANGLE_ROOT")) {
      cout << endl << "Printing old triangle" << endl;
      t->print(0);
   }
}
// close database file
op_mgr()->close_database("demo");
```

Notice that the above restore assumes proper installation of the Triangle persistent type in the type factory. Failing to use the type factory will cause an exception to be thrown in the restore_persistent call.

Polymorphic roots

Notice also that it is possible to restore an object as a base class type of the object actually stored, for example:

```
Shape* s = op_root<Shape>::restore("TRIANGLE_ROOT")
```

For this to work, `Shape` must be a base class of `Triangle`. An attempt to restore into an unrelated type, will cause an exception.

Persistent polymorphic pointers

Just like you have transient pointers pointing to objects in transient memory, in op_lite you have persistent pointers pointing to objects in the database. First an example of standard transient polymorphism in C++:

```
class Shape {
   virtual void display() = 0;
   …
};

class Circle : public Shape {
   virtual void display() { // draw circle }
   …
};

Shape* shape = new Circle();
shape->display();  // draws a circle
```

If Shape and Circle are op_lite persistent classes, can we achieve the same? Yes, we can

```
Shape* shape = op_new<Circle>();
shape->display();  // draws a circle
```

Then in a later session when we restore from the database

```
op_ptr<Shape> shape(pid);
shape->display();  // still draws a circle
```

In summary, a persistent pointer is represented using the template `op_ptr<T>` where T is the type of object being pointed to, or a base class of the concrete type. Therefore, op_ptr<T> can be used and considered as a standard polymorphic pointer, however persistent.

Ownership of objects

An `op_ptr<T>` is to be understood as a shared pointer that does not express ownership over the object being pointed to. The object pointed to actually resides in the database cache and is owned there.

Conversely, a `op_unique_ptr<T>` expresses such persistent ownership when it is being used as a member variable of another object. When the owning object is deleted from the database (using `op_delete`), then the objects being referred to via `op_unique_ptr<T>` will also be deleted from the database.

Persistent containers

One interesting feature in op_lite is the ability to define an STL (Standard Template Library) container as a standard persistent object data member.

In order to achieve this in a seamless manner, such containers are managed a little different than other variables, but this is not something an application writer needs to worry about, it "just works".

As seen from the application point of view, the developer simply uses STL-like container templates such as `op_vector<T>` and `op_map<T>`, and they can be used exactly as the ordinary standard containers, they even use the standard containers internally. *Note that only a subset of their interfaces have been exposed at the time of writing (it is trivial to expand the interface to be more complete, if needed).*

As seen from a low level database point of view, the container variables are standard BLOB (Binary Large Object) variables, which the database can handle without "understanding".

The "missing link" between the two views in this is being able to convert two ways between the real container and the BLOB representation in a portable manner. For this purpose, the library MessagePack /4/ is used. To put it simple, MessagePack is able to "pack" a container to a compact and portable representation that is stored in the database BLOB. When the data is restored, MessagePack does the "unpacking" from BLOB to container again. It all happens behind the scenes, and the application writer does not need to do anything special to make it work.

Persistent containers and polymorphic pointers

A particularly useful technique is to combine persistent containers with persistent polymorphic pointers as shown below. The *Shape* class may be understood as the base class of various concrete shapes (*Point, Circle, Line... etc*). By registering the concrete types in the type factory and using containers and pointers in this manner, a persistent container with true polymorphic pointers is achieved in a natural way and without much effort.

```
#include "Shape.h"
#include "op_lite/op_vector.h"

class POLY_SHAPES_PUBLIC ShapeCollection : public Shape {
public:
   ShapeCollection();
   virtual ~ShapeCollection();

   void op_layout(op_values& pvalues);
   void push_back(Shape* shape);
   void display();

private:
   op_vector<op_ptr<Shape>> m_shapes;  // container with polymorphic pointers
};
```

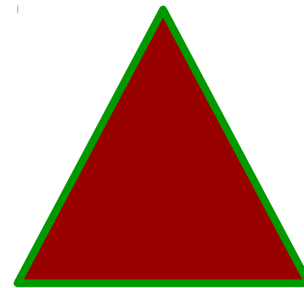
It is also possible to declare other user defined objects to be stored in containers in a similar fashion, see the MessagePack documentation. Strings are automatic. However, using persistent polymorphic pointers as shown above should cover most scenarios.

Examples

This chapter illustrates more complete example applications using op_lite.

Example "op_demo" – a persistent triangle

In this simple example, we shall implement and use persistent classes to represent triangles of various shapes in an op_lite database. The same job could be solved much simpler, but we have chosen to "overkill" the problem somewhat, for the purpose of illustrating various features of op_lite.

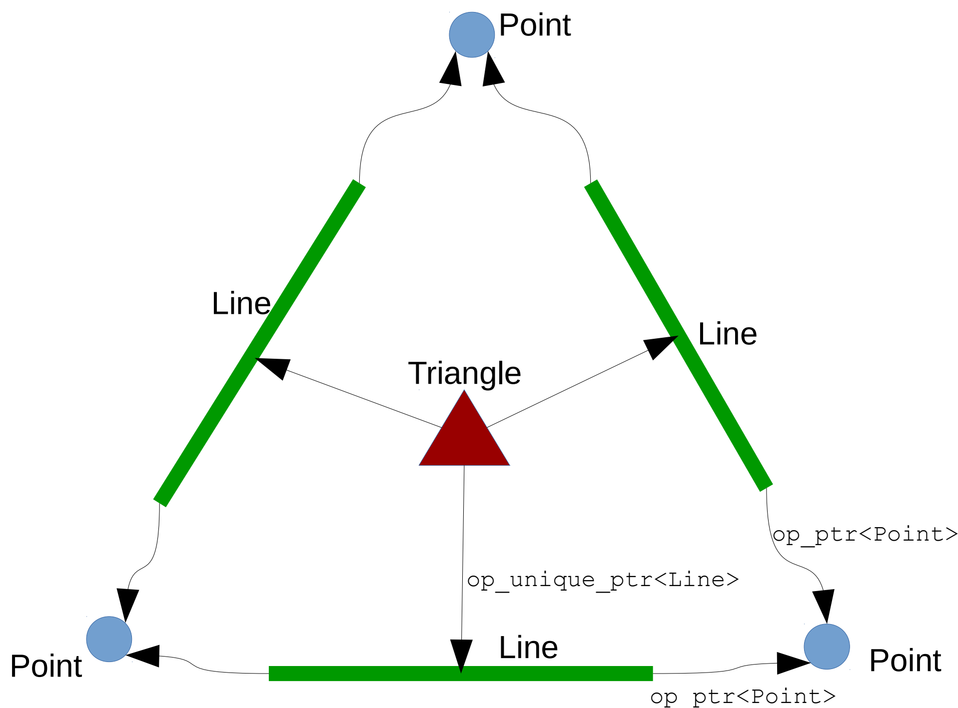


Drawing 1: A simple triangle

A typical example is illustrated at right. It is an area with 3 linear edges and 3 corner points.

In our example application we shall represent the triangle using 3 persistent classes: Point, Line and Triangle (we also need a transient op_demo class and a main program).

We shall also establish a kind of topology using pointers between instances of these classes. This is illustrated in the second drawing below:



Drawing 2: The triangle topology

Tre *Triangle* object owns 3 *Lines*. Each *Line* points to a *Point* in each end. Each corner *Point* is referenced by 2 *Lines*.

Point class, header

```
1  #ifndef POINT_H
2  #define POINT_H
3
4  #include "op_lite/op_object.h"
5
6  // op_lite demo: Persistent Point, defined by 2 persistent coordinates x and y
7
8  class Point : public op_object {
9  public:
10    // op_lite: default constructor is required
11    Point();
12
13    // construct point from coordinates
14    Point(double x, double y);
15    virtual ~Point();
16
17    // this is a required overload on all op_lite user objects
18    void op_layout(op_values& pvalues);
19
20    // return x and y coordinates
21    double x() const { return m_x; }
22    double y() const { return m_y; }
23
24    // print the Point starting at an offset from left
25    void print(int offset) const;
26
27 private:
28    op_double m_x;    // persistent double: x-coordinate
29    op_double m_y;    // persistent double: y-coordinate
30 };
31
32 #endif // POINT_H
33
```

Point class, implementation

```
1   #include "Point.h"
2   #include <iostream>
3   using namespace std;
4
5
6
7   Point::Point()
8   : op_construct(m_x)
9   , op_construct(m_y)
10  {}
11
12  Point::Point(double x, double y)
13  : op_construct_v1(m_x,x)
14  , op_construct_v1(m_y,y)
15  {}
16
17  Point::~Point()
18  {}
19
20  void Point::op_layout(op_values& pvalues)
21  {
22     op_bind(pvalues,m_x);
23     op_bind(pvalues,m_y);
24  }
25
26  void Point::print(int offset) const
27  {
28     for(int i=0;i<offset; i++) cout << ' ';
29     cout << "Point " << this << " = (" << x() << ", " << y() << ")" << endl;
30  }
31
```

Line class, header

```
1   #ifndef LINE_H
2   #define LINE_H
3
4   #include "op_lite/op_object.h"
5   #include "Point.h"
6   #include "op_lite/op_ptr.h"
7
8   class Line : public op_object {
9   public:
10     // op_lite: default constructor is required
11     Line();
12
13     // construct line from 2 persistent points
14     Line(Point* p1, Point* p2, const string& text);
15     virtual ~Line();
16
17     // this is a required overload on all op_lite user objects
18     void op_layout(op_values& pvalues);
19
20     // return pointers to Point
21     const Point* p1() const { return m_p1.get(); }
22     const Point* p2() const { return m_p2.get(); }
23
24     // print the Line starting at an offset from left
25     void print(int offset) const;
26
27  private:
28     // persistent pointers to Point
29     op_ptr<Point> m_p1;
30     op_ptr<Point> m_p2;
31
32     // persistent string member in this object
33     op_string     m_text;
34  };
35
36  #endif // LINE_H
```

Line class, implementation

```
 1  #include "Line.h"
 2  #include "Point.h"
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7
 8  Line::Line()
 9  : op_construct(m_p1)
10  , op_construct(m_p2)
11  , op_construct(m_text)
12  {}
13
14  Line::Line(Point* p1, Point* p2, const string& text)
15  : op_construct_v1(m_p1,p1)
16  , op_construct_v1(m_p2,p2)
17  , op_construct_v1(m_text,text)
18  {}
19
20  Line::~Line()
21  {}
22
23  void Line::op_layout(op_values& pvalues)
24  {
25     op_bind(pvalues,m_p1);
26     op_bind(pvalues,m_p2);
27     op_bind(pvalues,m_text);
28  }
29
30  void Line::print(int offset) const
31  {
32     for(int i=0;i<offset; i++) cout << ' ';
33     cout << "Line " << (string)m_text << endl;
34     offset +=3;
35     m_p1->print(offset);
36     m_p2->print(offset);
37  }
```

Triangle class, header

```
1   #ifndef TRIANGLE_H
2   #define TRIANGLE_H
3
4   #include "op_lite/op_object.h"
5   #include "op_lite/op_unique_ptr.h"
6   class Line;
7
8   // op_lite demo: Persistent triangle, defined by 3 persistent "Line" edges
9
10  class Triangle : public op_object {
11  public:
12
13     // op_lite: default constructor is required
14     Triangle();
15
16     // construct Triangle from lines
17     Triangle(Line* l1, Line* l2, Line* l3);
18     virtual ~Triangle();
19
20     // op_lite: "op_layout" is a required overload on all op_lite user objects
21     void op_layout(op_values& pvalues);
22
23     // return pointers to Line
24     const Line* l1() const { return m_l1.get(); }
25     const Line* l2() const { return m_l2.get(); }
26     const Line* l3() const { return m_l3.get(); }
27
28     // print the triangle starting at an offset from left
29     void print(int offset) const;
30
31  private:
32     // persistent pointers to lines
33     op_unique_ptr<Line> m_l1;
34     op_unique_ptr<Line> m_l2;
35     op_unique_ptr<Line> m_l3;
36  };
37
38  #endif // TRIANGLE_H
```

Triangle class, implementation

```
1  #include "Triangle.h"
2  #include "Line.h"
3
4  #include <iostream>
5  using namespace std;
6
7  Triangle::Triangle()
8  : op_construct(m_l1)
9  , op_construct(m_l2)
10 , op_construct(m_l3)
11 {}
12
13 Triangle::Triangle(Line* l1, Line* l2, Line* l3)
14 : op_construct_v1(m_l1,l1)
15 , op_construct_v1(m_l2,l2)
16 , op_construct_v1(m_l3,l3)
17 {}
18
19 Triangle::~Triangle()
20 {}
21
22 void Triangle::op_layout(op_values& pvalues)
23 {
24    op_bind(pvalues,m_l1);
25    op_bind(pvalues,m_l2);
26    op_bind(pvalues,m_l3);
27 }
28
29
30 void Triangle::print(int offset) const
31 {
32    for(int i=0;i<offset; i++) cout << ' ';
33    cout << "Triangle " << this << endl;
34    offset +=3;
35    m_l1->print(offset);
36    m_l2->print(offset);
37    m_l3->print(offset);
38 }
```

op_demo class, header

```
1   #ifndef OP_DEMO_H
2   #define OP_DEMO_H
3
4   #include <string>
5   using namespace std;
6
7   //  This is a very basic demo example of the op_lite database
8
9   class op_demo {
10  public:
11
12    // initialise type factory etc
13    static void init();
14
15    // create a new database and create a persistent triangle in it,
16    // then print it
17    static void create_database(const string& db_path);
18
19    // open an existing database and retore the persistent triangle in it,
20    // then print it, optinally delete triangle
21    static void read_database(const string& db_path, bool del_triang);
22  };
23
24  #endif // OP_DEMO_H
25
```

op_demo class, implementation

```
1   #include "op_demo.h"
2
3   #include "Point.h"
4   #include "Line.h"
5   #include "Triangle.h"
6
7   #include "op_lite/op_manager.h"
8   #include "op_lite/op_database.h"
9   #include "op_lite/op_root.h"
10  #include "op_lite/op_transaction.h"
11  #include "op_lite/op_new.h"
12
13  void op_demo::init()
14  {
15     // register the persistent types in the type factory
16     op_mgr()->type_factory()->install(new op_persistent_class<Point>());
17     op_mgr()->type_factory()->install(new op_persistent_class<Line>());
18     op_mgr()->type_factory()->install(new op_persistent_class<Triangle>());
19  }
20
```

Above is shown how the persistent classes are defined in the type factory

Then follows code for creating the database initially

```
21
22  void op_demo::create_database(const string& db_path)
23  {
24     cout << "op_demo::create_database" << endl;
25
26     if(op_database* db = op_mgr()->create_database("demo",db_path)) {
27        cout << "OK, created database " << db_path << endl;
28
29        // use a transaction to clean things up + it is more efficient
30        op_transaction trans(db);
31
32        // create 3 persistent Points
33        Point* pnt1  = op_new<Point>(-1.0, 0.0);
34        Point* pnt2  = op_new<Point>(+1.0, 0.0);
35        Point* pnt3  = op_new<Point>( 0.0,+1.0);
36
37        // create 3 persistent Lines, referring to above points
38        Line* l1     = op_new<Line>(pnt1,pnt2,"Line1");
39        Line* l2     = op_new<Line>(pnt2,pnt3,"Line2");
40        Line* l3     = op_new<Line>(pnt3,pnt1,"Line3");
41
42        // One persistent Triangle, referring to above lines
43        Triangle* t1 = op_new<Triangle>(l1,l2,l3);
44
45        // establish the Triangle as a root in the database
46        op_root<Triangle>* root = op_new<op_root<Triangle>>("TRIANGLE_ROOT",t1 );
47
48        cout << endl << "Printing new triangle" << endl;
49        t1->print(0);
50     }
51     else {
52        cout << "Could not create database " << db_path << endl;
53        return;
54     }
55
56     // finished. close database file
57     op_mgr()->close_database("demo");
58  }
59
60
61
```

Then follows code to read the database

```
61
62  void op_demo::read_database(const string& db_path, bool del_triang)
63  {
64     cout << "op_demo::read_database" << endl;
65
66     if(op_database* db = op_mgr()->open_database("demo",db_path)) {
67        cout << "OK, opened existing database " << db_path << endl;
68
69        // use a transaction to clean things up + it is more efficient
70        op_transaction trans(db);
71
72        // let us guess it is a triangle, and see if it so
73        if(Triangle* t = op_root<Triangle>::restore("TRIANGLE_ROOT")) {
74           cout << "OK, root object restored " << endl;
75
76           cout << endl << "Printing old triangle" << endl;
77           t->print(0);
78
79           if(del_triang) {
80              cout << endl << "Deleting Triangle " << t->pid().id() << endl;
81              op_delete(t);
82           }
83        }
84     }
85     else {
86        cout << "Could not open database " << db_path << endl;
87        return;
88     }
89
90     // close database file
91     op_mgr()->close_database("demo");
92  }
93
```

Main program

This main program uses wxWidgets for handling command line arguments, but this is irrelevant for the demo example as such. The code is shown here for completeness.

Se next page for the actual main program.

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  #include <stdexcept>
5  using namespace std;
6
7  // wxWidgets
8  #include <wx/defs.h>
9  #include <wx/platinfo.h>  // platform info
10 #include <wx/app.h>
11 #include <wx/string.h>   // wxString
12 #include <wx/cmdline.h>  // command line parser
13 typedef map<wxString,wxString> CmdLineMap;    // CmdLineMap
14
15 #include "op_demo.h"
16
17 static const wxCmdLineEntryDesc cmdLineDesc[] =
18 {
19 //   kind               shortName          longName           description                           parameterType          flag(s)
20   { wxCMD_LINE_SWITCH, wxT_2("h"),        wxT_2("help"),      wxT_2("command line parameter help"), wxCMD_LINE_VAL_NONE,   wxCMD_LINE_OPTION_HELP },
21   { wxCMD_LINE_PARAM,  wxT_2("db_path"),  wxT_2("db_path"),   wxT_2("<db_path>"),                   wxCMD_LINE_VAL_STRING, wxCMD_LINE_OPTION_MANDATORY  },
22   { wxCMD_LINE_SWITCH, wxT_2("c"),        wxT_2("c"),         wxT_2("create new database"),         wxCMD_LINE_VAL_STRING, wxCMD_LINE_PARAM_OPTIONAL  },
23   { wxCMD_LINE_SWITCH, wxT_2("d"),        wxT_2("d"),         wxT_2("delete Triangle in database"), wxCMD_LINE_VAL_STRING, wxCMD_LINE_PARAM_OPTIONAL  },
24   { wxCMD_LINE_NONE,   wxT_2(""),         wxT_2(""),          wxT_2(""),                            wxCMD_LINE_VAL_NONE,   wxCMD_LINE_PARAM_OPTIONAL  }
25 };
26
27 // helper function for command line parameters
28 void ParserToMap(wxCmdLineParser& parser, CmdLineMap& cmdMap)
29 {
30    size_t pcount = sizeof(cmdLineDesc)/sizeof(wxCmdLineEntryDesc) - 1;
31    for(size_t i=0; i<pcount; i++) {
32       wxString pname = cmdLineDesc[i].longName;
33       if(cmdLineDesc[i].kind == wxCMD_LINE_PARAM) {
34          cmdMap.insert(make_pair(pname,parser.GetParam(0)));
35       }
36       else {
37          // switch or option, mush check if present
38          if(parser.Found(pname)) {
39             wxString pvalue;
40             cmdMap.insert(make_pair(pname,pvalue));
41          }
42       }
43    }
44 }
45
```

The main program does a bit of command line massaging, then calls the relevant op_demo functions to do the real job.

Observe the use of exception handling, as op_light might throw an exception when serious errors are detected.

```
46  int main(int argc, char **argv)
47  {
48     // initialise wxWidgets library
49     wxInitializer initializer(argc,argv);
50
51     // parse command line
52     wxCmdLineParser parser(cmdLineDesc);
53     parser.SetSwitchChars(wxT("-"));
54     parser.SetCmdLine(argc,argv);
55     if(parser.Parse() != 0) {
56        // command line parameter error
57        return 1;
58     }
59     CmdLineMap cmdMap;
60     ParserToMap(parser,cmdMap);
61
62     // get the command line parameters
63     string db_path    = cmdMap["db_path"].ToStdString();
64     bool   db_create  = cmdMap.find("c") != cmdMap.end();
65     bool   del_tri    = cmdMap.find("d") != cmdMap.end();
66
67     try {
68        // make sure the type factory is always initialised
69        op_demo::init();
70
71        if(db_create) {
72           // create a new database
73           op_demo::create_database(db_path);
74        }
75        else {
76
77           // read an existing database
78           op_demo::read_database(db_path,del_tri);
79        }
80     }
81     catch( exception& e) {
82
83        // something unexpected happened
84        cout << "op_demo::Exception: " << e.what() << endl;
85     }
86
87     return 0;
88  }
89
```

The output example below shows 2 runs, creating and restoring the database. Observe that the transient pointer values have changed between the two runs, but the values and structure remain the same.

```
ca@Rustad5HP:/work/cpde3/acanve/opdb/op_demo/.cmp/gcc/bin/Release$
ca@Rustad5HP:/work/cpde3/acanve/opdb/op_demo/.cmp/gcc/bin/Release$ ./op_demo -c demo.db
op_demo::create_database
OK, created database demo.db

Printing new triangle
Triangle 0x1c3ab90
   Line Line1
      Point 0x1c39570 = (-1, 0)
      Point 0x1c37cf0 = (1, 0)
   Line Line2
      Point 0x1c37cf0 = (1, 0)
      Point 0x1c398c0 = (0, 1)
   Line Line3
      Point 0x1c398c0 = (0, 1)
      Point 0x1c39570 = (-1, 0)
ca@Rustad5HP:/work/cpde3/acanve/opdb/op_demo/.cmp/gcc/bin/Release$ ./op_demo  demo.db
op_demo::read_database
OK, opened existing database demo.db
OK, root object restored

Printing old triangle
Triangle 0x21d4530
   Line Line1
      Point 0x21d3530 = (-1, 0)
      Point 0x21d4af0 = (1, 0)
   Line Line2
      Point 0x21d4af0 = (1, 0)
      Point 0x21d4c80 = (0, 1)
   Line Line3
      Point 0x21d4c80 = (0, 1)
      Point 0x21d3530 = (-1, 0)
ca@Rustad5HP:/work/cpde3/acanve/opdb/op_demo/.cmp/gcc/bin/Release$ █
```

Building op_lite

There are 2 ways op_lite can built, either using the Code::Blocks project file, or via the provided makefiles generated from the Code::Blocks project file. Using the makefiles is the easiest option as they have been prepared to have few dependencies.

op_lite dependencies

Before building op_lite, its dependencies must be satisfied, i.e. you must download and build `boost` and `MessagePack`:

- boost library - http://www.boost.org/
  Boost is referenced from op_lite and must be built separately. Only 2 boost features are used
  - Bi-directional map, Boost.Bimap:
    http://www.boost.org/doc/libs/1_57_0/libs/bimap/doc/html/index.html
  - lexical_cast
    http://www.boost.org/doc/libs/1_57_0/doc/html/boost_lexical_cast.html

- MessagePack library - http://msgpack.org/
  This library is used for realising persistence of containers such as op_vector and op_map. As for boost, this library is referenced from op_lite and must be built separately. Build-instructions for Windows and Linux are found at
  https://github.com/msgpack/msgpack-c/blob/master/QUICKSTART-C.md

- op_lite also contains 2 source subfolders “lzmalib” and “sqlite3”, which are compiled together with op_lite.

Building with 'Makefile.msvc' on Windows

The file 'Makefile.msvc' builds op_lite on Windows using MS Visual Studio 2010 (Express or full edition). To use the makefile, open the *“Visual Studio Command Prompt (2010)”* from the Windows start menu, navigate to the op_lite source directory. Edit the file `Makefile.msvc` and adjust the two lines on the top, so they point to where `boost` and `MessagePack` have been installed and built.

```
MSGPACK_INCLUDE = E:\\cpde3\\zdep\\3rdparty\\msgpack\\msgpack-c\\include
BOOST_INCLUDE   = E:\\cpde3\\zdep\\3rdparty\\boost\\boost_1_55_0
```

Then run the makefile

```
$ nmake -f Makefile.msvc
```

The generated `op_lite.lib` and `op_lite.dll` files are found in the `.cmp\msvc\bin\Release` subfolder.

Building with 'Makefile' on Linux

The file 'Makefile' builds op_lite under Linux using g++. To use the makefile, open a terminal window in the op_lite source directory. Edit the file `Makefile` and adjust the two lines on the top, so they point to where `boost` and `MessagePack` have been installed and built.

```
MSGPACK_INCLUDE = /usr/local
BOOST_INCLUDE   = /home/ca/home_work/cpde_root/zdep/3rdparty/boost/boost_1_55_0/
```

Then run the makefile

$ `make`

The generated `libop_lite.so` shared object library is found in the `.cmp\gcc\bin\Release` subfolder.

References

/1/ **ObjectStore PSE Pro for C++**
http://www.dei.isep.ipp.pt/~alex/publico/ostore/userguide1.html
PSE for C++ and PSE Pro for C++ are lightweight persistent storage engines that allow you to store and retrieve objects in their native C++ format. Without PSE, you could store C++ data in flat files or relational databases, and write code to perform I/O and translation to and from object format. But such code is complex, error prone, and difficult to maintain. PSE Pro eliminate these problems in favor of

- A simple to use and easy to learn API
- A high-performance virtual memory mapping architecture

/2/ **SQLite**
http://www.sqlite.org/

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.

/3/ **C++ Object Persistence with ODB**
http://www.codesynthesis.com/products/odb/doc/manual.xhtml
ODB is an object-relational mapping (ORM) system for the C++ programming language.

/4/ **MessagePack**
MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves.
http://msgpack.org/