The Reposit Project

An Improved Solution
For Autogenerating
QuantLibXL Source Code
Father Guido Sarducci's Five Minute University

In five minutes, you learn what the average college graduate remembers five years after he or she is out of school.

https://www.youtube.com/watch?v=kO8x8eoU3L4

Reposit Project Five Second University:

- Replace the gensrc Python script with the reposet SWIG module
- QuantLibAddin object wrapper code autogenerated not handwritten
- Objective: Export all of QuantLib to Excel
Reposit Project Website

http://www.quantlib.org/reposit

Overview

Reposit implements a repository in which objects may be instantiated, queried, updated, and destroyed, facilitating deployment of object oriented functionality to procedural platforms such as spreadsheets.

The project also provides a utility to parse the header files of a C++ library and autogenerate the source code for a corresponding Excel addin. This makes it easy to export the functionality of a C++ library to Excel.

QuantLib

Reposit is a subproject of the QuantLib C++ analytics library and shares the QuantLib project structure with regard to distribution and licensing. The mailing list for reposist is quantlib-users@lists.sf.net. The list archives may be searched here.
Documentation

http://quantlib.org/reposit/documentation.html
Documentation for the Reposit project.

docs/ObjectHandler-docs-1.5.0.html/index.html
Documentation for the ObjectHandler repository.

docs/swig/Reposit.html
Documentation for the SWIG module.
Overview

ObjectHandler
- Object repository
- Object base class

QuantLibObjects
- Classes which inherit from Object and wrap QuantLib
- Native support for serialization

QuantLibAddin
- Functional interface which exports QuantLibObjects to target platforms (C++, Excel)

gensrc (deprecated)
- autogenerates addin source code

SWIG reposit module
- autogenerates object wrapper and addin source code

QuantLib
namespace QuantLib {
    class Instrument {
        /*...*/
    }
    class Swap : public Instrument {
        /*...*/
    }
}

QuantLibObjects
namespace QuantLibObjects {
    class Instrument : public ObjectHandler::LibraryObject
    {
        /*...*/
    }
    class Swap : public Instrument {
        /*...*/
    }
}

ObjectHandler namespace ObjectHandler {
    map<string, Object*> repository;
    class Object {
        /*...*/
    }
    template <class T>
    class LibraryObject : public Object
    {
        /*...*/
    }
}

std::string idSwap = qlSwap(/*...*/);
qlInstrumentSetPricingEngine(/*...*/);
std::cout << "swap PV = " << qlInstrumentNPV(idVanillaOption);

QuantLibAddin – C++
namespace QuantLibAddinCpp {
    qlInstrumentNpv();
    qlSwap();
}

Excel Workbook

C++ Client

function metadata
gensrc
source code generation
SWIG reposit module
source code generation
SWIG interface files

QuantLibXL
namespace QuantLibXL {
    qlInstrumentNpv();
    qlSwap();
}
This page provides an overview of how ObjectHandler, QuantLibAddin, and QuantLibXL will change after gensrc is replaced by the Reposit SWIG module.

<table>
<thead>
<tr>
<th>Component</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code generation</td>
<td>• The gensrc Python script is discontinued and is replaced by the Reposit SWIG module.</td>
</tr>
<tr>
<td>ObjectHandler</td>
<td>• Some ObjectHandler source code that was previously autogenerated by gensrc is now maintained manually.</td>
</tr>
<tr>
<td></td>
<td>• Otherwise no changes to ObjectHandler code or functionality.</td>
</tr>
<tr>
<td></td>
<td>• I might like to rename ObjectHandler to Reposit.</td>
</tr>
<tr>
<td>QuantLibAddin</td>
<td>• Object wrapper source code that was previously handwritten is now autogenerated.</td>
</tr>
<tr>
<td></td>
<td>• Some less important source code (e.g. enumerations) that was previously autogenerated is now maintained manually.</td>
</tr>
<tr>
<td></td>
<td>• C++ Addin is now easier to use and its interface is now more similar both to QuantLib and to QuantLibXL.</td>
</tr>
<tr>
<td></td>
<td>• Conversion/Coercion code completely rewritten, cleaned up, clarified, and commented. Many other minor improvements.</td>
</tr>
<tr>
<td>QuantLibXL</td>
<td>• Old design supports 1,000+ functions, new design currently supports only a dozen or so functions, enough to price an Equity Option.</td>
</tr>
<tr>
<td></td>
<td>• It is hoped that the new design will be easier to use and will result in more QuantLib functionality being exported to Excel.</td>
</tr>
<tr>
<td></td>
<td>• In principle, changing the method of autogenerated source code should not change the design of QuantLibXL. In practice, some things will change, e.g. function names.</td>
</tr>
</tbody>
</table>
Used in the normal way, SWIG performs two steps:

1) parse the SWIG interface files
2) generate a single source code file which can be compiled into an addin for the target platform.

QuantLib-SWIG uses SWIG in the usual way:
Reposit relies on the core SWIG functionality to parse the interface files. Reposit then does its own thing for code generation. The standard SWIG output file is generated, but it is not used. Instead Reposit generates a completely different set of output files.

We will describe the Reposit output files in more detail. But first let us answer The Most Frequently Asked Question...
SWIG Interface Files

How Come Reposit Doesn’t Reuse QuantLib’s SWIG Interface Files?

QuantLib

```cpp
// plain option and engines
%
using QuantLib::VanillaOption;
typedef boost::shared_ptr<Instrument> VanillaOptionPtr;
%

rename(VanillaOption) VanillaOptionPtr;
class VanillaOptionPtr : public boost::shared_ptr<Instrument> {
  public:
    %extend {
      VanillaOptionPtr(
        const boost::shared_ptr<Payoff>& payoff,
        const boost::shared_ptr<Exercise>& exercise) {
          boost::shared_ptr<StrikedTypePayoff> stPayoff =
            boost::dynamic_pointer_cast<StrikedTypePayoff>(payoff);
          QL_REQUIRE(stPayoff, "wrong payoff given");
          return new VanillaOptionPtr(new VanillaOption(stPayoff,exercise));
        }
    }
};
}
```

Reposit

```cpp
namespace QuantLib {
  class Instrument {
    public:
      //Instrument();
      void setPricingEngine(const boost::shared_ptr<QuantLib::PricingEngine>& engine);
      QuantLib::Real NPV();
  };
  class VanillaOption : public Instrument {
    public:
      VanillaOption(const boost::shared_ptr<QuantLib::StrikedTypePayoff>& payoff,
                    const boost::shared_ptr<QuantLib::Exercise>& exercise);
  }
}
```

Shown at left:

- the QuantLib SWIG interface file for an Option
- the Reposit SWIG interface file for an Option

The QuantLib SWIG files were written before SWIG introduced support for boost shared pointers. The file contains additional logic to hide the shared pointer.

Reposit’s SWIG interface file is much more similar to the corresponding QuantLib C++ header file.
# Output Files

ReposiT generates six output files global to the Addin:

<table>
<thead>
<tr>
<th>Path</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComplexLibAddin/clo/obj_all.hpp</td>
<td><code>#include directives</code></td>
</tr>
<tr>
<td>ComplexLibAddin/clo/serialization/register_creators.cpp</td>
<td>register addin classes with the serialization layer</td>
</tr>
<tr>
<td>ComplexLibAddin/clo/serialization/create/create_all.hpp</td>
<td><code>#includes relating to creation of serialization objects</code></td>
</tr>
<tr>
<td>ComplexLibAddin/clo/serialization/register/serialization_register.hpp</td>
<td><code>#includes relating to registration for serialization</code></td>
</tr>
<tr>
<td>ComplexLibAddin/clo/serialization/register/serialization_all.hpp</td>
<td><code>#includes relating to registration for serialization</code></td>
</tr>
<tr>
<td>ComplexLibAddin/AddinCpp/add_all.hpp</td>
<td><code>#includes for the C++ addin</code></td>
</tr>
</tbody>
</table>

ReposiT generates six output files for each group of functions (instruments, term structures, etc):

<table>
<thead>
<tr>
<th>Component</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ComplexLibAddin/clo/valueobjects/vo_xx.?pp</td>
<td>implementation of value objects in support of serialization</td>
</tr>
<tr>
<td>ComplexLibAddin/clo/serialization/create/create_xx.?pp</td>
<td>functions to create objects as they are deserialized</td>
</tr>
<tr>
<td>ComplexLibAddin/clo/serialization/register/serialization_xx.?pp</td>
<td>register addin classes with the serialization layer</td>
</tr>
<tr>
<td>ComplexLibAddin/clo/obj_xx.?pp</td>
<td>addin objects that wrap classes in the library</td>
</tr>
<tr>
<td>ComplexLibAddin/AddinCpp/add_xx.?pp</td>
<td>the functions in the C++ addin</td>
</tr>
<tr>
<td>ComplexLibXL/clxl/functions/function_xxx.cpp</td>
<td>The functions in the Excel addin</td>
</tr>
</tbody>
</table>
SimpleLib

Very nearly* the smallest Reposit project that it is possible to have.

1. Define your Library

```cpp
namespace SimpleLib {
    std::string func();
    class Adder {
    private:
        long x_; // private data member
    public:
        Adder(long x) : x_(x) {} // constructor
        long add(long y) { return x_ + y; } // public method
    };
};
```

2. Create your SWIG interface file

```swig
%typemap(rp_tm_xll_cod) SimpleLib::Adder * "C";
%module(Adder)
%feature("rp:group", "adder");
%feature("rp:obj_include")
```

3. Generate your Addins

```swig```

* you could make it smaller by dropping the class and keeping only the function...
ComplexLib

This example project supports a bucket list of all features supported by Reposit.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description/Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>std::string helloWorld();</td>
</tr>
<tr>
<td>Typedefs</td>
<td>typedef double Real;</td>
</tr>
<tr>
<td>Objects</td>
<td>class Foo { ... };</td>
</tr>
<tr>
<td>Inheritance</td>
<td>class Bar : public Foo { ... };</td>
</tr>
<tr>
<td>Conversions</td>
<td>void f(Real r);</td>
</tr>
<tr>
<td>Coercions</td>
<td>void setQuote(X x); // x could be a double or a string id of a Quote object</td>
</tr>
<tr>
<td>Enumerated Types</td>
<td>enum AccountType { Current, Savings };</td>
</tr>
<tr>
<td>Enumerated Classes</td>
<td>class TimeZoneUtc : public TimeZone { /* ... */ };</td>
</tr>
<tr>
<td>Enumerated Pairs*</td>
<td>template&lt;type A, type B&gt; class Foo { ... };</td>
</tr>
<tr>
<td>Custom Enumerations*</td>
<td>Calendar factory – create new joint calendars on the fly as they are named.</td>
</tr>
<tr>
<td>Overrides</td>
<td>The developer may suppress autogeneration of selected source code files in order to provide handwritten code.</td>
</tr>
<tr>
<td>Serialization*</td>
<td>Serialization of objects, exactly as in the old build of ObjectHandler/QuantLibAddin/QuantLibXL.</td>
</tr>
</tbody>
</table>

* not yet supported
Inheritance

Example – Step 1 of 7 – Overview

Here we take one of the features supported by Reposit – Inheritance – and work through the ComplexLib example step by step.

When your C++ library (e.g. QuantLib) contains inheritance relationships, the code to be autogenerated by Reposit for each class will differ depending upon whether the class has a parent and/or a constructor.

<table>
<thead>
<tr>
<th>Parent?</th>
<th>Constructor?</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>full class inheriting LibraryObject</td>
<td>If the library class is a base class, and if it has a constructor, then reposit autogenerates a complete implementation of the wrapper class. For base class ComplexLib::Foo, you get a wrapper class ComplexLibAddin::Foo which inherits from helper class ObjectHandler::LibraryObject.</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>OH_LIB_CLASS</td>
<td>If the library class is a base class, and if it has no constructor, reposit still generates a wrapper class. But the wrapper is a skeleton and the entire implementation is provided by macro OH_LIB_CLASS.</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>full class inheriting Object</td>
<td>If the library class is a derived class, and if it has a constructor, then reposit autogenerates a complete implementation of the wrapper class. For base class ComplexLib::Bar deriving from ComplexLib::Foo, you get a wrapper class ComplexLibAddin::Bar deriving from ComplexLibAddin::Foo.</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>OH_OBJ_CLASS</td>
<td>If the library class is a derived class, and if it has no constructor, reposit still generates a wrapper class. But the wrapper is a skeleton and the entire implementation is provided by macro OH_OBJ_CLASS.</td>
</tr>
</tbody>
</table>
Inheritance

Example – Step 2 of 7 – Library Header File

#ifndef complex_lib_inheritance_hpp
#define complex_lib_inheritance_hpp

// Test inheritance and polymorphism.
#include <string>

namespace ComplexLib {

    // One base class, one derived.
    class Base {
    public:
        virtual std::string f() { return "ComplexLib::Base::f()"; }
        virtual ~Base() {}
    };

    class Derived : public Base {
    public:
        virtual std::string f() { return "ComplexLib::Derived::f()"; }
    };

    // Hierarchy of 3 classes.
    class A {
    public:
        virtual std::string f0() = 0;
        virtual ~A() {}
    };

    class B : public A {
    public:
        virtual std::string f1() = 0;
    };

    class C : public B {
    public:
        virtual std::string f0() { return "ComplexLib::C::f0()"; }
        virtual std::string f1() { return "ComplexLib::C::f1()"; }
    };
};

#endif
This is a SWIG interface file, written for consumption by the Reposit SWIG module.

This file defines the subset of the C++ header file that we want to export to our Addins (C++ and Excel).

This file is very similar in format to the corresponding C++ header file.
Example – Step 4 of 7 – Autogenerated Object Wrapper Code

This is the autogenerated wrapper code.

In this example we call it ComplexLibAddin, in the real world this would be QuantLibAddin (QuantLibObjects).

Each class here inherits from ObjectHandler::Object and holds a pointer to a ComplexLib object.
Inheritance

Example – Step 5 of 7 – Autogenerated Addin Code

This is the autogenerated code for the C++ and Excel addins.

As Excel worksheet functions cannot directly handle C++ constructors, this code is functional, not object oriented.

All of the code required for the necessary datatype conversions has been autogenerated.
#include <iostream>
#include "AddinCpp/add_all.hpp"
#include "oh/addin.hpp"
#include "test_all.hpp"

#ifdef TEST_INHERITANCE

void testInheritance() {
    std::cout << std::endl;
    std::cout << "Testing inheritance" << std::endl;
    std::cout << std::endl;

    ComplexLibAddinCpp::clBase("base");
    std::cout << ComplexLibAddinCpp::clBaseF("base") << std::endl;
    ComplexLibAddinCpp::clDerived("derived");
    std::cout << ComplexLibAddinCpp::clBaseF("derived") << std::endl;
    std::cout << ComplexLibAddinCpp::clDerivedF("derived") << std::endl;
    try {
        std::cout << ComplexLibAddinCpp::clDerivedF("base") << std::endl;
    } catch(const std::exception &e) {
        std::cout << "Expected error : " << e.what() << std::endl;
    }

    ComplexLibAddinCpp::clC("c");
    std::cout << ComplexLibAddinCpp::clAF0("c") << std::endl;
    std::cout << ComplexLibAddinCpp::clBF1("c") << std::endl;
}
#endif

For C++, we write by hand some code to test the Addin.

For Excel we enter the same formulas into a workbook (see below).
Inheritance

Example – Step 7 of 7 – Client Code / Spreadsheets

This is the output from the C++ client program, and from the corresponding test workbook.

On both platforms the interface and behavior is the same.
Improved C++ Addin

QuantLibAddin interface is now more similar both to QuantLib and to QuantLibXL.
Development Environment

Reposit SWIG module
repos/reposit/swig/Source/Modules/reposit.cxx

Reposit SWIG interface file
repos/reposit/swig/Lib/reposit/reposit.swg

SimpleLib Example
repos/reposit/swig/Examples/reposit/simple

ComplexLib Example
repos/reposit/swig/Examples/reposit/complex

new QuantLibAddin
repos/reposit/quantlib/QuantLibAddin2

new QuantLibXL
repos/reposit/quantlib/QuantLibXL2
Typemaps

Reposit defines a series of typemaps. Each typemap is used to generate the required code at a specific point in a source code file.

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Typemap</th>
</tr>
</thead>
<tbody>
<tr>
<td>rp_val_*</td>
<td>rp_tm_val_prm</td>
</tr>
<tr>
<td>rp_val_*</td>
<td>rp_tm_val_dcl</td>
</tr>
<tr>
<td>rp_val_*</td>
<td>rp_tm_val_ser</td>
</tr>
<tr>
<td>rp_val_*</td>
<td>rp_tm_val_nam</td>
</tr>
<tr>
<td>rp_val_*</td>
<td>rp_tm_valIni</td>
</tr>
<tr>
<td>rp_val_*</td>
<td>rp_tm_val_cnv</td>
</tr>
<tr>
<td>rp_ser_*</td>
<td>rp_tm_crv_cnv</td>
</tr>
<tr>
<td>rp_obj_*</td>
<td>rp_tm_obj_ret</td>
</tr>
<tr>
<td>rp_obj_*</td>
<td>rp_tm_obj_rdc</td>
</tr>
<tr>
<td>rp_add_*</td>
<td>rp_tm_add_ret</td>
</tr>
<tr>
<td>rp_add_*</td>
<td>rp_tm_add_prm</td>
</tr>
<tr>
<td>rp_add_*</td>
<td>rp_tm_add_cnv</td>
</tr>
<tr>
<td>rp_add_*</td>
<td>rp_tm_add_cll</td>
</tr>
<tr>
<td>rp_add_*</td>
<td>rp_add_ret</td>
</tr>
<tr>
<td>rp_add_*</td>
<td>rp_tm_add_oh_get</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_tm_xll_cod</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_tm_xll_prm</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_tm_xll_cnv</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_tm_xll_cll_obj</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_tm_xll_cll_val</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_tm_xll_ret</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_xll_get</td>
</tr>
<tr>
<td>rp_xll_*</td>
<td>rp_tm_xll_rdc</td>
</tr>
</tbody>
</table>

Normally SWIG typemaps are applied directly to native C++ types, e.g. bool, double, etc.

Reposit instead defines a few placeholders for C++ types. Each addin must map its own types to these placeholders.

```plaintext
%apply rp_tp_double { LongDouble };
%apply const rp_tp_double & { const LongDouble & };
%apply rp_tp_cnv { Grade };
%apply rp_tp_crc { Grade2 };
%apply rp_tp_enm { AccountType };
%apply rp_tp_enm { Account2::Type2 };
%apply rp_tp_enm_cls { boost::shared_ptr<TimeZone> };
```

The application developer has to map the types defined in his library to the type placeholders defined by Reposit. This will be the most difficult step for exporting QuantLib to QuantLibXL.
Status

Done:
• Working prototype supporting an Equity Option, including addins for C++ and Excel.

To Do:
• Implement support for the rest of the QuantLib functionality – Yield curve bootstrap, price interest rate swap, everything else.
• Implement support for serialization
• For all addin functions, need to autogenerate the trigger/permanent/anonymous parameters
• LibreOffice Calc addin?