Rcpp: R/C++ Interface Classes

Using C++ Libraries from R

Version 1.0

Dominick Samperi

January 2006

Abstract

A set of C++ classes that facilitate the process of using C++ libraries (like **QuantLib**) from within the **R** statistical software system is described.

1 Introduction

The **R** system is written in the C language, and it provides a C API for package developers who have typically coded functions to be called from **R** in C or FORTRAN. **Rcpp** provides C++ classes that make it relatively easy to use C++ libraries from **R**.

The **Rcpp** "bare bones" approach is to find a small set of data structures that can be easily passed between **R** and C++ in a language-natural way (on both the **R** and the C++ side), and that is sufficient for the problem domain under study. Technical details having to do with **R** API internals are hidden from the **Rcpp** user.[1] Since the author's focus was on applications to finance the choice of data structures was somewhat biased, but it can be extended without much effort.

2 Quick Start Guide

To run the sample function `RcppExample`, simply install the **Rcpp** package in the usual way, and run

```
> library(Rcpp)
> example(RcppExample)
```

[1] This is done in a style similar to the JDBC Java database interface, so it makes the C++ code look like a "smart database."

There is a binary version of **Rcpp** for Windows. When it is installed it defines the function `RcppExample` and places all of the source code into RHOME/library/Rcpp/doc. To build from source under Windows you will have to configure an **R** development environment.[2] At the very least you will need the MinGW compiler (or Dev-Cpp) and the UNIX tools (see previous footnote).

Under Linux everything should happen automatically. The configure script `configure.in` (together with `autoconf`) sets up the environment under Linux, and the script `configure.win` is used under Windows.

The source file for the function `RcppExample` is `RcppExample.cpp`, located in the `doc` subdirectory of the installed **Rcpp** package (or in the subdirectory `inst/doc` of the uninstalled source directory). The source files for the **Rcpp** classes are `Rcpp.cpp` and `Rcpp.hpp`, located in the same place. The example makes calls to a trivial C++ test library named `Mylib`. The source files used to build this library can also be found in the `doc` directory.

To add your own C++ code (for testing), unpack the **Rcpp** source archive (the `tar.gz` file), insert your C++ source files into `Rcpp/src`, insert **R** source files that make calls to your C++ code into `Rcpp/R`, and build (from above the **Rcpp** directory):

```
$ R CMD INSTALL --library Rcpp.test Rcpp
```

Test your code in **R** using:

```
> library(Rcpp, lib.loc='Rcpp.test')
> myfunc()
```

We do not cover the details of building a shared library and packaging it for use by **R**. For this information see the document "Writing R Extensions" available at the **R** Web site: http://cran.r-project.org. Of course, when you create your own package you must change **Rcpp** to your package name, and modify the information in `DESCRIPTION`. If you need to link with other C++ libraries you will have to write a custom makefile (or modify `configure.in` and use `autoconf`).

If you want to use the `QuantLib` C++ library talk with Dirk Eddelbuettel about making your code part of the `RQuantLib` package. It currently uses **Rcpp** to expose `QuantLib` fixed-income functions.

3 Important Note

It is important to remember that there is a potential for conflicts when two **R** packages use the same C++ library (whether or not this is done with the help of **Rcpp**). For example, if two **R** packages use `QuantLib`, and if both packages are used at the same time, then the static (singleton)

[2]This consists of: **R**, the UNIX tools for **R** from http:/www.murdoch-sutherland.com/Rtools, the MinGW GNU compiler, ActivePerl from http://www.activestate.com, MikTeX (TeX for Windows), and Microsoft's HTML help tool.

classes of `QuantLib` may not be manipulated properly: what singleton object gets modified will depend on the order in which the packages are loaded!

4 Assumptions

We assume that four kinds of objects will be passed between **R** and C++. On the **R** side they include the following:

1. A list of named values of possibly different types
2. A list of named values of numeric type (real or integer)
3. A numeric vector
4. A numeric matrix

An example of the first kind of object would be constructed using the **R** code

```
params <- list(method = "BFGS", someDate = c(10,6,2005))
```

The allowed types are `character`, `real`, `integer`, and `vector` (of length 3, holding a date in the form: month, day, year). Note that support for the corresponding `Date` type on the C++ side depends on **QuantLib** and is not available when **Rcpp** is used without **QuantLib**. In this case a dummy `Date` class is compiled in that knows only how to print itself.[3]

An example of the second kind of object is

```
prices <- list(ibm = 80.50, hp = 53.64, c = 45.41)
```

Here all values must be numeric.

Finally, examples of the last two kinds of objects are:

```
vec <- c(1, 2, 3, 4, 5)
mat <- matrix(seq(1,20),4,5)
```

Objects of the first kind are called parameter lists and are managed using the class `RcppParams` (see below), while objects of the second kind are called named lists and are managed using the class `RcppNamedList`. Objects of the third kind are managed by `RcppVector<type>` template classes, and objects of the last kind are managed by `RcppMatrix<type>` template classes.

5 User Guide

To call a C++ function named `MyFunc`, say, the **R** code would look like:

```
.Call("MyFunc", p1, p2, p3)
```

[3]There are many C++ date classes available on the Internet, but unfortunately, there is no C++ standard date class.

where the parameters (can be more or less than three, of course) can be objects of the kind discussed in the previous section. Usually this call is made from an intermediate **R** function so the interactive call would look like

```
>   MyFunc(p1, p2, p3)
```

Now let us consider the following code designed to make a call to a C++ function named `RcppSample`

```
params <- list(method = "BFGS", tolerance = 1.0e-8, startVal = 10)
a <- matrix(seq(1,20), 4, 5)
.Call("RcppSample", params, a)
```

The corresponding C++ source code for the function `RcppSample` using the **Rcpp** interface and protocol might look like the code in Figure 1.

```
#include "Rcpp.hpp"
RcppExtern SEXP RcppSample(SEXP params, SEXP a) {
    SEXP rl=0; // return list to be filled in below
    try {
        RcppParams rp(params);
        string name = rp.getStringValue("method");
        double tolerance = rp.getDoubleValue("tolerance");
        ...
        RcppMatrix<double> mat(a);
        // Use 2D matrix via mat(i,j) in the usual way
        ...
        RcppResultSet rs;
        rs.add("name1", result1);
        rs.add("name2", result2);
        ...
        rs.add("params", params, false);
        rl = rs.getResultList();
    } catch(std::exception& ex) {
        error("Exception:  %s\n", ex.what());
    }
    catch(...)  {
        error("Exception:  unknown reason\n");
    }
    return rl;
}
```

Figure 1: Rcpp use pattern.

Here `RcppExtern` ensures that the function is callable from **R**. The SEXP type is an internal type used by **R** to represent everything (in particular, our parameter values and the return value). It can be quite tricky to work with SEXP's directly, and thanks to **Rcpp** this is not necessary.

Note that all of the work is done inside of a `try/catch` block. Exception messages generated by the C++ code are propagated back to the **R** user naturally (even though **R** is not written in C++).

The first object created is of type `RcppParams` and it encapsulates the `params` SEXP. Values are extracted from this object naturally as illustrated here. There are `getTypeValue(name)` methods for `Type` equal to `Double`, `Int`, `Bool`, `String`, and `Date`.

Rcpp checks that the named value is present and that it has the correct type, and returns an error message to the **R** user otherwise. Similarly, the other encapsulation classes described below check that the underlying **R** data structures have the correct type (this eliminates the need for a great deal of checking in the **R** code that ultimately calls the C++ function).

The matrix parameter `a` is encapsulated by the `mat` object of type `RcppMatrix<double>` (matrix of double's). It could also have been encapsulated inside of a matrix of int's type, in which case non-integer values would be truncated toward zero. Note that SEXP parameters are read-only, but that these encapsulating classes work on a copy of the original, so they can be modified in the usual way:

```
mat(i,j) = whatever
```

The `RcppVector<type>` classes work similarly.

In these matrix/vector representations subscripting is range checked. It is possible to get a C/C++ style (unchecked) array copy of an `RcppMatrix` and `RcppVector` object by using the methods `cMatrix()` and `cVector()`, respectively. The first method returns a pointer of type `type **`, and the second returns a pointer of type `type *` (where `type` can be `double` or `int`). These pointer-based representations might be useful when matrices/vectors need to be passed to software that does not know about the **Rcpp** classes. No attempt should be made to free the memory pointed to by these pointers as it is managed by **R** (it will be freed automatically when `.Call` returns).

Returning to the example, we see that after `mat` is constructed, C++ classes would typically be used to do some computations (not shown here), and when they complete an object of type `RcppResultSet` is used to construct the list (SEXP) to be returned to **R**. Results to be returned are added to the list using the `add` method where the first parameter is the name that will be seen by the **R** user. The second parameter is the corresponding value—it can be of type `double`, `int`, `string`, `RcppMatrix<double>`, etc.

The last call to `add` here is used to return the input SEXP parameter `params` as the last output result (named "params"). The boolean flag `false` here means that the SEXP has not been protected. This will be the case unless the SEXP has been allocated by the user (not an input parameter).

For a concrete example see `RcppExample.cpp`. For examples employing `QuantLib` see the files `discount.cpp` and `bermudan.cpp` from the `RQuantLib` package.

6 Quick Reference

In this quick reference "`type`" can be `double` or `int`.

`RcppParams` constructor and methods

```
RcppParams::RcppParams(SEXP)
double RcppParams::getDoubleValue(string)
int RcppParams::getIntValue(string)
string RcppParams::getStringValue(string)
bool RcppParams::getBoolValue(string)
Date RcppParams::getDateValue(string)
```

`RcppNamedList` constructor and methods

```
RcppNamedList::RcppNamedList(SEXP)
int RcppNamedList::getLength()
string RcppNamedList::getName(int)
double RcppNamedList::getValue(int)
```

Matrix and vector constructors

```
RcppMatrix<type>(SEXP a)
RcppMatrix<type>(int nrow, int ncol)
RcppVector<type>(SEXP a)
RcppVector<type>(int len)
```

Matrix and vector methods

```
type& RcppMatrix<type>::operator(int i, int j)
type& RcppVector<type>::operator(int i)
type* RcppVector<type>::cVector()
type** RcppMatrix<type>::cMatrix()
```

`RcppResultSet` constructor and methods

```
RcppResultSet::RcppResultSet()
void RcppResultSet::add(string,double)
void RcppResultSet::add(string,int)
void RcppResultSet::add(string,string)
void RcppResultSet::add(string,double*,int)
void RcppResultSet::add(string,double**,int,int)
void RcppResultSet::add(string,int*,int)
void RcppResultSet::add(string,int**,int,int)
void RcppResultSet::add(string,RcppVector<type>&)
void RcppResultSet::add(string,RcppMatrix<type>&)
void RcppResultSet::add(string,SEXP,bool)
```

The last method here is provided for users who want work with SEXP's directly, or when the user wants to pass one of the input SEXP's back as a return value, as we did in the example above. The boolean flag tells **Rcpp** whether or not the SEXP provided has been protected.

A SEXP that is allocated by the user may be garbage collected by **R** at any time so it needs to be protected using the PROTECT function to prevent this. A SEXP that is passed to a C++ function by **R** does not need to be protected because **R** knows that it is in use.