Building **R** Packages that call **C++** Functions

Rcpp: R/C++ Interface Classes Version 5.0

Dominick Samperi*

September 20, 2006

Abstract

A set of C++ classes that facilitates the process of using C++ with the Open Source **R** statistical software system is described. The intent is to make it easier for researchers and practitioners to share their ideas by providing working prototypes that others can experiment with, test, and contribute to. The powerful visualization capabilities of **R** adds an important dimension to the model selection and testing process. This is particularly important in fields like applied finance where abstract formal analysis must be complemented with clinical studies and tractability and robustness tests. The classes described here make it easier to use existing C++ libraries (like **QuantLib**), or to rapidly develop an **R** interface to C++ software that implements new modeling ideas. This is done with the help of mappings between **R** and C++ types like vectors, matrices, dates, and **R** data frames. Support for calling **R** functions from C++ is also included.

1 Note to Users of Previous Versions

The class `RcppNamedList` has been renamed to `RcppNumList` (because only numeric data is permitted). There are several new classes including `RcppDate`, `RcppFrame`, and `RcppFunction` (the last class can be used to call **R** functions from C++).

There are new package functions named `RcppTemplateDemo` and `RcppTemplateVersion`, and `RcppTemplate` now uses a `NAMESPACE` file. See below for details.

2 Overview

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. The contribution of **Rcpp** is to add C++ classes that make it relatively easy to use C++ objects and libraries from **R**. Conversely, functions defined on the **R** side can be called from C++.

The **Rcpp** design strategy 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 most problem domains. The data structures currently supported include heterogeneous parameter lists (where you would pass in doubles, reals, strings, etc., with names attached), homogeneous parameter lists (where all parameter values are numeric and named), 1D vectors, 2D matrices, Dates, **R**-style data frames (similar to data base tables), and **R** functions.

Technical details having to do with **R** API internals are hidden from the **Rcpp** user. Of course, low-level **R** API features can still be used to support objects (like matrices of dimension greater than two) that are not currently supported by the **Rcpp** API.

*Email: dsamperi@DecisionSynergy.com. I am grateful for helpful comments from Dirk Eddelbuettel, Hin-Tak Leung, Uwe Ligges, Duncan Murdoch, Brian Ripley, and Paul Roebuck.

One limitation of **Rcpp** is that it does not address the problem of persistence, that is, having C++ objects maintain their state between function calls. When a call to a C++ function returns all objects created during the call are destroyed. The work-around is to use **R** to maintain state, and recreate the C++ objects as needed during the calls. The overhead of the latter is typically much smaller than the overhead of doing compute-intensive work in **R** itself (**R** is an interpreted language). It is possible to pass large arrays to C++ functions by address, thus eliminating the need to copy this data for every call, but this should be avoided where possible since it is inconsistent with pass-by-value semantics of **R**.

Example **R** packages that currently use **Rcpp** include `FinancePack`, and `RQuantLib`. The `CreditPricer` function in `FinancePack` illustrates how to use the underlying **R** API to pass arrays by address to C++ functions.

The **Rcpp** class library is shipped as part of the `RcppTemplate` package, available from the official **R** web site, http://cran.r-project.org. The official reference on writing **R** extensions is "Writing R Extensions," available from the same site. It contains some comments on how to use foreign languages like C and FORTRAN, but says little about C++. The latter document should be consulted for details about the **R** API that we omit below.

The **R** package `RcppTemplate` can be used as a template for building **R** packages that use C++ class libraries. It includes a working sample function `RcppExample` that illustrates how to use **Rcpp**. To run the sample function install the `RcppTemplate` package in the usual way, and use:

```
> library(RcppTemplate)
> example(RcppExample)
```

There is a documentation page for the example that can be viewed with:

```
> ?RcppExample
```

The source code and man page for the example can be found in the source archive (the .tar.gz file, not the Windows binary .zip file). To extract the source archive change directory to a convenient location and use the command line:

```
$ tar -xvzf RcppTemplate_VVV.tar.gz
```

Under Windows the 'tar' command is part of the Rtools package (see Appendix A or Appendix B for details). Here 'VVV' stands for the package version number.

At this point you will be above the root of the package directory hierarchy (above RcppTemplate). The C++ source code for RcppExample can be found in RcppTemplate/src, and the R code and man pages can be found in RcppTemplate/R and RcppTemplate/man, respectively. The contents of RcppExample.cpp will be easier to understand after reading the rest of this document.

It should be clear now that building a package requires some familiarity with UNIX style command-line tools. **R** can be viewed as a UNIX application that has been ported to Windows with the help of a collection of tools that help to make Windows look like UNIX. A minimal UNIX-like environment is defined for Windows by modifying your path as follows (under Windows):

```
$ set path=c:\Rtools\bin;%path% (modify path as needed)
```

This defines UNIX commands like `tar`, `cp`, `rm`, etc. For more information about Rtools and the Windows package build process see Appendix A (GNU compiler) or Appendix B (MS Visual C++ compiler).

There are a couple of differences between UNIX and Windows that need to be kept in mind. Under UNIX most tools expect path names to have the directory names separated by a forward slash (`/`), whereas under Windows the command-line tools expect a backward slash separator (\). The tools in Rtools have been modified to accept separator slashes in either direction.

One common problem that occurs when Rtools is used under Windows is that the wrong version of a tool like `find` is found during the build process because the user's PATH variable is not set properly (either Windows `find`, or cygwin `find` may be found). To fix this problem make sure that all of the **R** development tools appear early on the search path (see the way path was set above), or use a command file (.bat file) that explicitly sets the path before running **R** commands under Windows.

Another UNIX/Windows issue has to do with blank spaces in file names. It is generally not a good idea to build **R** packages in directories with names that contain spaces (like "My Documents" under Windows). This will probably cause the build process to fail.

Note that packages must be submitted to CRAN in source format (.tar.gz file), and CRAN does not support Visual C++, so the GNU compiler must be used for a CRAN submission. On the other hand, Visual C++ can be used for testing and benchmarking, for example.

Having extracted the RcppTemplate source archive we next sketch how to test and build an **R** package in a generic (operating system-independent) way. For more details about Windows see the appendices.

A test version of the package (with no customization) can be created by changing directory so that you are above the package root (RcppTemplate) and using:

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

The code can then be run from the RcppTemplate.test directory as follows:

```
> library(RcppTemplate, lib.loc='RcppTemplate.test')
> example(RcppExample)
```

Before we begin customizing the package it will simplify things if we delete the vignette file containing the documentation for RcppTemplate:

```
$ rm RcppTemplate/inst/doc/RcppAPI.Rnw
```

It is now possible to insert your C++ source files into `RcppTemplate/src`, and insert **R** source files into `RcppTemplate/R` that make calls to your C++ code (using the `.Call` interface). Follow the pattern in `RcppExample.cpp` and `RcppExample.R`. The build procedure will automatically find and compile source files in the `src` directory, so you do not need to create a Makefile (this applies to UNIX, you have to modify Makefile.win under Windows, unless you are using MSVC). To add documentation files follow the pattern in `RcppExample.Rd` (see "Writing R Extensions" for more details).

After you have tested a few functions to get the general idea (without changing the package name), you can rename the package as desired by changing: the information in DESCRIPTION, the name of the package root directory, the string 'RcppTemplate' to your package name in the R code, and in the man pages.

You can link against your own C++ libraries by following the pattern used to link against the **Rcpp** library, or you can link against external libraries. For an example of the latter case, look at the `RQuantLib` package. It uses **Rcpp** and links against the `QuantLib` and `Boost` class libraries.

The configure file `configure.in` can be modified as desired to check for libraries that may be needed to build your package (UNIX only). Remember to run `autoconf` after modifying it.

If you named your package MyPackage, then you can install it on your machine in the standard location using (assuming you are above MyPackage, and logged in as root if on UNIX):

```
$ R CMD INSTALL MyPackage
```

To build a source archive that can be submitted to CRAN, first make sure that it passes check (no need to be the root user to do this):

```
$ R CMD check MyPackage
```

If all is well, then you can make a source archive (.tar.gz file) for submission using:

```
$ R CMD build MyPackage
```

3 The Package Demo Function

The file `R/zzz.R` normally contains initialization code. It defines a function named `.First.lib` that **R** calls to initialize the package and load its shared library (when one exists).

`RcppTemplate` has been configured to define another function in this file named `RcppTemplateDemo`. There is also an associated documentation page. This function can be used to run any demos that have been placed into the package `demo` subdirectory. Such demos are just files containing **R** code with the extension ".R". Each demo should have a corresponding short description in the file named `00Index` in the same directory. Look at the `demo` subdirectory of `RcppTemplate` to see how things are arranged.

To run all of the demos for the package simply type

```
> RcppTemplateDemo()
```

You will be presented with a list of demos to choose from. Entering zero will cause the contents of `00Index` to be displayed. Entering anything else besides a demo number will terminate `RcppTemplateDemo`.

A good way to introduce new users to your package is to provide a few demos in this way. Note that users need no knowledge about your package to run the demos. Of course, your demo function would be named `YourPackageNameDemo`.

4 The Package Version Function

The `RcppTemplate` package includes a function named `RcppTemplateVersion` that displays the version number of the package that is currently installed, along with the version number of **Rcpp** that was used to build the package. When configuring your own package this function (and its man page) should be modified by replacing all instances of the string 'RcppTemplate' with 'YourPackageName'. In particular, the function should be named `YourPackageNameVersion`.

Note that the version number shown is for the *installed* package, not a test version that you might be using from a local directory. If the package has not been installed into the standard place running this function will result in an error message about the package not being installed.

During the installation process the **Rcpp** library `libRcpp.a` is built in `RcppTemplate/RcppSrc` (where the source code for the library is located), and object files created in `RcppTemplate/src` are linked against this library in order to create the package shared library `RcppTemplate.so` (DLL under Windows). The `RcppSrc` directory is not part of the installed package (you will only find it in the source archive).

The only visible trace of **Rcpp** that is left behind after the package is installed (or after a Windows binary .zip file is created) is the **Rcpp** license file `LICENSE-Rcpp.txt`. This file describes terms of use, and also keeps track of the version number of **Rcpp** that was used to build your package. It is used by `YourPackageNameVersion` to display version information.

In the process of customizing RcppTemplate for your use the part of the build procedure that saves the license file `LICENSE-Rcpp.txt` to the package root directory should be retained. Alternatively, you can simply copy this file manually from the `RcppSrc` directory to the `inst` directory of your source archive (**R** places files in `inst` into the root directory of the installed package as part of the install process).

5 Package Namespaces

Package namespaces are used to prevent conflicts when two or more packages use the same name for a function. Maintaining a package `NAMESPACE` file is also a good way to keep track of all of the functions that are available. There are other uses for the `NAMESPACE` file—see *Writing R Extensions* for more information.

The `RcppTemplate` package includes a simple `NAMESPACE` file that declares that it will be using the `RcppTemplate` shared library. It also exports all of the function names that will be visible to the package user: `RcppExample`, `RcppTemplateDemo`, `RcppTemplateVersion`, and `print.RcppExample`. Look at the `NAMESPACE` file (in the package root directory) to see how this is done.

When two packages export a function of the same name, the one that is selected is the one that was defined most recently (in the package that was attached last). To override this behavior it is possible to specify which version should be used as follows

```
> pkg::name()
```

This says to use the version defined in package `pkg`.

When the second package is attached (using the `library` command) **R** will issue a warning that the name defined in this package is masking the one that was previously defined (the masked name is still accessible, as we have just seen).

6 Controlling R's Output

The **R** language supports a rich collection of object-oriented features like inheritance and polymorphism. For our purposes we will use one very simple feature in order to control what gets printed when a variable name assigned to is entered on a line by itself. By default this invokes a generic `print` function that displays every value, even in deeply nested lists. If you are returning a large matrix this is probably not the desired behavior.

The file `RcppExample.R` illustrates how to work around this problem by writing a customized `print` function for the returned value. The code first assigns the value returned by `.Call` to the variable `val`. Then it assigns a class name to this variable, and writes a specialized `print` function for this class. This means if you enter this variable name on a line by itself the function `print.Classname` is called instead of `print`. What is used for `Classname` must not conflict with class names already in use, and in the present case `RcppExample` is used.

Incidentally, **R** supports multiple inheritance since a vector of class names can be assigned to objects in this way.

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

8 Assumptions

We assume that the following kinds of objects will be passed between **R** and C++ (the C++ class used to manage each object type is shown in parentheses):

1. A heterogeneous list of named values of possibly different types (`RcppParams`),

2. A homogeneous list of named values of numeric type (`RcppNumList`)
3. A numeric 1D vector (`RcppVector<type>`)
4. A numeric 2D matrix (`RcppMatrix<type>`)
5. A Date (`RcppDate`)
6. A data frame (`RcppFrame`)
7. A vector of Dates (`RcppDateVector`)
8. A vector of strings (`RcppStringVector`)
9. An R function (`RcppFunction`)

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

```
params <- list(method = "BFGS", tol=1.0e-8, maxiter=1000)
```

The allowed types are `character`, `real`, `integer`, and `Date`.

The latter is **R**'s date class (does not include time of day). On the C++ side a Date is represented by an `RcppDate`, a basic date class that keeps track of month/day/year and Julian day number, and supports a few basic operations like subtraction, incrementing, and comparing dates.[1]

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.

Examples of vector and matrix are:

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

R style Date's can be passed as one of the parameters in a heterogeneous list, or they can be passed as a vector of dates created like this, for example:

```
dateStrings <- c('2006-7-1', '2010-2-3', '2015-7-1')
dates <- as.Date(dateStrings, '%Y-%m-%d')
```

See `RcppExample.cpp` for an example.

An example data frame is:

```
df <- data.frame(id=c(1,2,3),fac=c('weak','strong','moderate'),
        answer=c(TRUE,FALSE,TRUE))
```

[1] When the compiler flag `USING_QUANTLIB` is set an implicit cast from `RcppDate` to **QuantLib** `Date` is defined, and these operations are disabled because they confuse the compiler. They can also be disabled by unsetting the flag `RCPP_DATE_OPS` (see `Rcpp.hpp`). The internal date representation used by `RcppDate` is the number of days since Monday, January 1, 4713BC (the so-called Julian day number), while the internal representation used by **R** is the number of days since January 1, 1970.

Vector and matrix objects are managed by the template classes `RcppVector<type>` and `RcppMatrix<type>`, where `type` can be `double` or `int`.

R functions can be called from C++ with the help of the `RcppFunction` class. This is done by subclassing and using utility functions defined in the superclass to build lists or vectors to be passed to the **R** function. The function can return any kind of **R** object, in theory, but in most applications it will return a SEXP represenation for a real number or a vector.[2] The use of `RcppFunction` is an advanced topic. Examples of its use are contained in `RcppExample.cpp`.

See `RcppExample.cpp` for sample code using these classes. See the **Rcpp** Quick Reference (Appendix C) for a complete list of classes and methods.

9 User Guide

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

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

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.

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:

[2] It is not a good idea for the function to hold onto PROTECT-ed **R** storage, even though it will automatically UNPROTECT this storage when it is destroyed (on return to **R**), because repeated calls to the function may overflow **R**'s protection stack. To avoid this problem copy any PROTECTED **R** objects to C++ objects to be returned, and UNPROTECT the **R** objects.

```
#include "Rcpp.hpp"
RcppExtern SEXP RcppSample(SEXP params, SEXP a) {
    SEXP rl=R_NilValue; // Return this when there is nothing to return.
    char* exceptionMesg=NULL;
    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) {
        exceptionMesg = copyMessageToR(ex.what());
    }
    catch(...)  {
        exceptionMesg = copyMessageToR("unknown reason");
    }
    if(exceptionMesg != NULL)
        error(exceptionMesg);
    return rl;
}
```

Figure 1: Use pattern for **Rcpp**.

```
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 after `.Call` returns).

An STL `vector` copy of an `RcppVector` object can be obtained by using the `stlVector` method of the `RcppVector` class. An STL matrix, or `vector<vector<type> >`, copy of an `RcppMatrix` object can be obtained by using the `stlMatrix` method of the `RcppMatrix` class. See `RcppExample.cpp` for examples.

Returning to the example, we see that the `mat` and `vec` parameters are used to construct `RcppVector` and `RcppMatrix` objects, respectively. These would typically be used to do some computations (not shown here). When the computations are finished an object of type `RcppResultSet` is constructed that contains the data values 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`, `vector<double>`, `vector<string>` `vector<vector<double> >`, `RcppMatrix<double>`, `RcppFrame`, 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 examples employing `QuantLib` see the files `discount.cpp` and `bermudan.cpp` from the `RQuantLib` package.

To use data frames, simply pass the data frame like we passed a vector or matrix **R** object above. If the SEXP parameter corresponding to the data frame is named `df`, then a C++ code fragment that uses it might look like Figure 2.

```
RcppFrame frame(df);
vector<vector<ColDatum> > table = frame.getTableData();
int nrow = table.size();
int ncol = table[0].size(); // Get ncols from first row.
for(int row=0; row < nrow; row++) {
    for(int col=0; col < ncol; col++) {
        if(table[row][col].getType() == COLTYPE_FACTOR) {
            level = table[row][col].getFactorLevel();
            string name = table[row][col].getFactorLevelName();
        }
    }
}
```

Figure 2: Using data frames with **Rcpp**.

Here an **R** data frame is represented in C++ as a vector of rows, each of which is a vector of columns of type `ColDatum`, and the data that each `ColDatum` contains can be one of the following supported column types:

```
COLTYPE_DOUBLE, COLTYPE_INT, COLTYPE_STRING,
COLTYPE_FACTOR, COLTYPE_LOGICAL, and COLTYPE_DATE.
```

There are associated methods getType(), getDoubleValue(), getIntValue(), etc.

Setter methods are also available for use when you are creating an RcppFrame object to be returned to **R**. These include setDoubleValue(double x), setFactorValue(string *names, int numLevels, int level), etc. It is the user's responsibility to ensure that columns added in this way have consistent types from one row to the next—**Rcpp** will throw an exception if an inconsistency is detected.

In the current implementation the vector of level names provided to setFactorValue() is stored with every factor value, and these name vectors must be consistent across rows (each factor in the same row must have the same vector of level names—see `RcppExample.cpp`). The problem of optimizing this design by factoring out the level names is left for a future release.

For an example of how to construct a new RcppFrame object to be returned to **R** see RcppExample.cpp. The object that is returned is actually a "pre-data frame," because it is not recognized by **R** as a data frame, but it is a simple matter to turn it into a data frame. For example, RcppExample.cpp returns an RcppFrame object in `result$PreDF`. It can be turned into a data frame using:

```
df <- data.frame(result$PreDF)
```

10 Appendix A: Using GNU MinGW under Windows

In this section we explain how to use the GNU C++ compiler under Windows to build a dynamic link library, and how to build a binary **R** package that uses it. The GNU C++ compiler can be downloaded in the form of the MinGW package for Windows, or the Dev-Cpp front-end (a graphical user interface built on top of GNU C++). Section 2 is a prerequisite for this appendix.

In the following the package name will be RcppTemplate. You can simply use this package name and add source files and R functions as needed. Later when you see how everything fits together you can change the package name everywhere. This will involve changes to DESCRIPTION, package root directory, the R files, and the man pages. The procedure as as follows.

1. (**Download and Install**) Download and install the necessary tools. This includes

   - the UNIX tools for **R** from http://www.murdoch-sutherland.com/Rtools,
   - the MinGW GNU compiler (or Dev-Cpp),
   - ActivePerl from http://www.activestate.com,
   - MikTeX (TeX for Windows),
   - Microsoft's HTML help tool.

   The HTML help tool can be downloaded from Microsoft—see the murdoch-sutherland site for more information. Under Windows NT4 (and some versions of Windows 2000) you will need to install a patched version of `ld.exe`, available at http://www.murdoch-sutherland.com/Rtools.

   As explained in Section 2, make sure that the Rtools UNIX-like tools are in your search path:

```
$ set path=c:\Rtools\bin;%path% (modify as needed)
```

2. (**Prepare package source**)

   Extract the RcppTemplate source archive (the .tar.gz file) into some convenient location (in your private space, not in the **R** installation directory). Use the `tar` command that comes with Rtools (here 'VVV' stands for the version number):

```
$ cd <some convenient place>
$ tar -xvzf RcppTemplate_VVV.tar.gz
```

After issuing this command you will be located in a directory directly above the package root directory (RcppTemplate).

3. (**Build the binary R package**) Check the Windows batch file (or command file)

```
RcppTemplate\inst\doc\MakeWinBin.bat
```

to make sure that it points to the correct places (where the tools like Rtools and R have been installed). Then change directory so that you are above the package root directory (RcppTemplate), copy the batch file to the same location, and run the batch file:

```
$ MakeWinBin RcppTemplate
```

If everything was installed properly this should compile everything, make the DLL, and create the package binary (.zip file).

4. (**Prepare source for customization**) So far we have simply built the binary version of RcppTemplate that can be downloaded from CRAN. To customize the package it will be helpful to delete the vignette file (this simplifies the build process):

```
$ rm RcppTemplate\inst\doc\RcppAPI.Rnw
```

5. (**Customizing**) As we explained in Section 2 it is now possible to add source files to `RcppTemplate\src`, and **R** files to `RcppTemplate\R`. The file `RcppTemplate\src\Makefile.win` must be updated to include any new source files that you create. Unlike the generic case, it is convenient to use Windows command files (.bat files) to drive the testing, build, and release process, like we did in Step 3 above. Note that under Windows commands like 'R CMD build' can be replaced with 'Rcmd build'. When you are familiar with the way packages are structured you can rename the package by making the appropriate changes as described previously.

11 Appendix B: Using Microsoft Visual C++

In this section we explain how to build a dynamic link library (DLL) using the Microsoft Visual C++ Express IDE (part of Visual Studio 2005). It is also called MSVC, or MSVC 8.0. We also explain how to build a binary **R** package that uses this DLL. Note that this package can be used for internal testing and benchmarking only (cannot be uploaded to CRAN) because CRAN does not support Visual C++, and will not accept binary submissions. Section 2 is a prerequisite for this appendix.

1. (**Download and Install**) Download and install the necessary tools. This includes

- the UNIX tools for **R** from http://www.murdoch-sutherland.com/Rtools,
- the MinGW-utils tools from http://www.mingw.org,
- ActivePerl from http://www.activestate.com,
- MikTeX (TeX for Windows),
- Microsoft's HTML help tool.

It is not necessary to download the entire MinGW compiler, only the binary version of the tools is needed. Of course, you must also have installed **R** and MS Visual C++. The HTML help tool can be downloaded from Microsoft—see the murdoch-sutherland site for more information.

Make sure that the UNIX tools from Rtools and the GNU C++ MinGW utilities are in your path:

```
$ set path=c:\Rtools\bin;%path% (modify as needed)
$ set path=c:\MinGW\bin;%path% (modify as needed)
```

Also be sure that the MSVC command-line tools are in your environment by working from the terminal window that is provided by MSVC (this is a separate application, not part of the IDE).

2. (**Build interface library**) Export symbols from the R.dll file and make a library interface file that Visual C++ can use. This is done where **R** is installed as follows:

```
$ cd C:\Program Files\R\R-2.3.1\bin (modify as required)
$ pexports R.dll > R.exp
$ lib /def:R.exp /out:Rdll.lib
```

Here `lib` is the library command that comes with Visual C++, and `pexports` is part of the MinGW-utils package.

3. (**Prepare package source**) Extract the RcppTemplate source archive (the .tar.gz file) into some convenient location (in your private space, not in the **R** installation directory). Use the `tar` command that comes with Rtools (here 'VVV' stands for the version number):

```
$ cd <some convenient place>
$ tar -xvzf RcppTemplate_VVV.tar.gz
```

After issuing this command you will be located in a directory directly above the package root directory (RcppTemplate).

By default RcppTemplate is designed to be compiled with MinGW (GNU compiler). In order to build with Visual C++, we make a package subdirectory named `RcppTemplate\MSVC\RcppTemplate`, and we copy the source files from `RcppTemplate\src` to this directory. We also have to delete the original source directory, along with the GNU configuration file. Finally, we need to make a directory that will hold the DLL file that we are about to build:

```
$ mkdir RcppTemplate\MSVC
$ mkdir RcppTemplate\MSVC\RcppTemplate
$ cp RcppTemplate\src\RcppExample.cpp RcppTemplate\MSVC\RcppTemplate
$ rm -rf RcppTemplate\src RcppTemplate\configure.win
$ mkdir RcppTemplate\inst\libs
```

To simplify the build process it will also be useful to delete the vignette file for RcppTemplate:

```
$ rm RcppTemplate\inst\doc\RcppAPI.Rnw
```

4. (**Build DLL using MSVC**) Start the Visual C++ IDE, and select File / New Project. In the New Project dialog box set the project name to RcppTemplate, and the location to

```
C:\RcppTemplate\MSVC (modify as needed)
```

Make sure the 'create directory for solution' box is not checked, and select the Win32 Console Application template. Under Application Settings, turn precompiled headers off. When you make a new project like this the following files are created: stdafx.cpp, stdafx.h, RcppTemplate.cpp. All three of them should be deleted because they will not be used (and they can cause problems if present).

Next we add source files to the project. Select Project / Add Existing Item, and add RcppExample.cpp. Then use the same command and navigate to the directory containing the **Rcpp** source files (RcppSrc) and add the files Rcpp.cpp and Rcpp.hpp in turn. Later you can add files of your own design in the same way.

Set global options as follows (indentation corresponds to MSVC menu levels):

```
Tools
    Options
        Projects and Solutions
            VC++ Directories
                Include dirs:  C:\Program Files\R\R-2.3.1\include
                Library dirs:  C:\Program Files\R\R-2.3.1\bin
```

Modify the paths here as needed.

Next we set project-specific options. These need to be set separately for Debug and Release (optimized) builds. Let's set the mode to Release, and cover the options for this case.

```
Project
    Properties
        General
            Configuration type:  Dynamic library (.dll)
            Use of MFC: Use MFC in a static library
        C/C++
            General
                Additional include dirs:  ..\..\RcppSrc (add others if neeeded)
            Preprocessor
                Preprocessor Defs:  BUILDING_DLL (add to options already present)
        Linker
            Input
                Additional dependencies:  Rdll.lib (add others if needed)
```

Now that everything is configured we can build the dynamic link library (DLL) by selecting: Build / Build Solution. If everything goes well this will create:

```
RcppTemplate\MSVC\RcppTemplate\Release\RcppTemplate.dll
```

Be sure to exit the IDE before moving on to the next step (otherwise the **R** build process may try to delete files that the IDE has locked).

5. (**Create a binary R package**) Move the DLL file from the place where MSVC puts it to

```
RcppTemplate\inst\libs\RcppTemplate.dll
```

Check the Windows batch file `RcppTemplate\inst\doc\MakeWinBin.bat` to make sure that it points to the correct places (where the tools like Rtools and **R** have been installed). Then change directory so that you are above your working package root directory (RcppTemplate), copy the batch file to the same location, and run the batch file:

```
$ MakeWinBin RcppTemplate
```

This should create RcppTemplate_VVV.zip, a binary **R** package file that can be installed under Windows in the usual way.

6. (**Customizing**) Unlike the standard situation discussed in Section 2, here customized source files should be placed into

`RcppTemplate\MSVC\RcppTemplate`

and added to the project using Project / Add Existing Item, as explained above. There is of course no need to work with a Makefile when using MSVC. It is convenient to use Windows command files (.bat files) to drive the testing, build, and release process, like we did in the previous step. Note that under Windows commands like 'R CMD build' can be replaced with 'Rcmd build'. When you are familiar with the way packages are structured you can rename the package by making the appropriate changes as described previously.

12 Appendix C: Rcpp 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)
RcppDate RcppParams::getDateValue(string)
```

RcppNumList constructor and methods
```
RcppNumList::RcppNumList(SEXP)
int RcppNumList::size()
string RcppNumList::getName(int)
double RcppNumList::getValue(int)
```

RcppDate constructors, methods, and friends
```
RcppDate::RcppDate() [defaults to 1/1/1970]
RcppDate::RcppDate(int month, int day, int year)
RcppDate::RcppDate(int Rjdn) [Rjdn = 0 for 1/1/1970]
int RcppDate::getMonth()
int RcppDate::getDay()
int RcppDate::getYear()
int RcppDate::getJDN()
friend RcppDate operator+(const RcppDate&, int offset)
friend int operator-(const RcppDate&, const RcppDate&)
friend bool operator<(const RcppDate&, const RcppDate&)
friend bool operator>(const RcppDate&, const RcppDate&)
friend bool operator<=(const RcppDate&, const RcppDate&)
friend bool operator>=(const RcppDate&, const RcppDate&)
friend bool operator==(const RcppDate&, const RcppDate&)
RcppDate::RcppDate(Date date) [requires QuantLib]
Date RcppDate::operator Date() [requires QuantLib]
```

Matrix and vector constructors
```
RcppMatrix<type>(SEXP a)
RcppMatrix<type>(int nrow, int ncol)
RcppVector<type>(SEXP a)
RcppVector<type>(int len)
RcppStringVector(SEXP sv)
RcppDateVector(SEXP dv)
```

Matrix and vector methods

```
int RcppVector<type>::size()
int RcppStringVector::size()
int RcppDateVector::size()
string& RcppStringVector::operator()(int i)
RcppDate& RcppDateVector::operator()(int i)
type& RcppMatrix<type>::operator()(int i, int j)
type& RcppVector<type>::operator()(int i)
vector<type> RcppVector<type>::stlVector()
vector<vector<type> > RcppMatrix<type>::stlMatrix()
type* RcppVector<type>::cVector()
type** RcppMatrix<type>::cMatrix()
```

RcppFrame constructors and methods

```
RcppFrame::RcppFrame(SEXP df) [input from R]
RcppFrame::RcppFrame(vector<string> colNames) [user created]
vector<string>& RcppFrame::getColNames()
vector<vector<ColDatum> >& RcppFrame::getTableData()
void RcppFrame::addRow(vector<ColDatum> rowData)
```

ColDatum constructor and methods

```
ColDatum::ColDatum()
ColType ColDatum::getType()
int ColDatum::getIntValue()
double ColDatum::getDoubleValue()
int ColDatum::getLogicalValue()
string ColDatum::getStringValue()
RcppDate ColDatum::getDateValue()
int ColDatum::getFactorLevel()
string ColDatum::getFactorLevelName() [name for this level]
string *ColDatum::getFactorLevelNames() [all level names]
int ColDatum::getFactorNumLevels()
double ColDatum::getDateRCode()
void ColDatum::setIntValue(int val)
void ColDatum::setDoubleValue(double val)
void ColDatum::setLogicalValue(int val)
void ColDatum::setStringValue(string val)
void ColDatum::setDateValue(RcppDate date)
void ColDatum::setFactorValue(string *names, int numNames, int level)
```

ColType values

```
COLTYPE_DOUBLE
COLTYPE_INT
COLTYPE_LOGICAL
COLTYPE_STRING
COLTYPE_FACTOR
COLTYPE_DATE
```

`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,RcppDate&)
void RcppResultSet::add(string,RcppDateVector&)
void RcppResultSet::add(string,RcppStringVector&)
void RcppResultSet::add(string,vector<type>&)
void RcppResultSet::add(string,vector<vector<type> >&)
void RcppResultSet::add(string,vector<string>&)
void RcppResultSet::add(string,RcppVector<type>&)
void RcppResultSet::add(string,RcppMatrix<type>&)
void RcppResultSet::add(string,RcppFrame&)
void RcppResultSet::add(string,SEXP,bool)
```

RcppFunction constructors and methods

```
RcppFunction::RcppFunction(SEXP fn) [input from R]
void RcppFunction::setRVector(vector<double>& v)
void RcppFunction::setRListSize(int size)
void RcppFunction::appendToRList(string name, double val)
void RcppFunction::appendToRList(string name, int val)
void RcppFunction::appendToRList(string name, string val)
void RcppFunction::appendToRList(string name, RcppDate& val)
void RcppFunction::clearProtectionStack()
SEXP RcppFunction::vectorCall()
SEXP RcppFunction::listCall()
```

The last method in `RcppResultSet` is provided for users who want to 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.

The last class `RcppFunction` provides an interface for calling functions defined on the R side. The user must subclass and define adapter interfaces that use utility functions in the superclass to make calls to **R** functions. The functions can be called with list or vector parameters (using `listCall()` and `vectorCall()`, respectively). See `RcppExample.cpp` and `RcppExample.Rd` for examples.