Rcpp: A Class Library for R/C++ Object Mapping and R Package Development

Dominick Samperi*

October 24, 2009

Abstract

A set of C++ classes that facilitates the process of using C++ with the Open Source **R** statistical software system is described. The C++ classes described here model **R** data structures like vectors, matrices, factors, dates, data frames, time series, and functions. Objects of these types can be instantiated from pointers (or SEXP's) that **R** passes to the C++ side. The C++ model objects can also be constructed from C++ data structures and later returned to **R** in a format that it recognizes. The function model can be used to make calls from C++ to **R** functions that take parameters (and return values) of any of the types just mentioned.

1 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. A commonly used interface (the so-called `.C` interface) permits only simple objects like vectors and matrices to be passed as parameters (and returned). A somewhat newer interface called the `.Call` interface can be used to pass pointers to **R** objects to the C/C++ side, but this interface has not yet been fully exploited to provide an object mapping between **R** and C++.

The purpose of the **Rcpp** class library is to provide such a mapping. At present it provides a mapping between some of the most commonly used data structures on the **R** side to C++ classes. These data structures include vectors, matrices, factors, dates, data frames, time series, and functions. It is possible to call C++ functions from **R** with arguments of any of these types, and C++ objects can call **R** functions with arguments of these types as well. Functions on either side can return values of any of the supported types. Currently the **R** types supported are all S3 classes.

There are several interface packages available that enable **R** to communicate with other language systems including Java, Perl and Python. An important goal of these packages is to expand the **R** user community by making **R** multi-lingual. The goal of Rcpp is somewhat different. Here we want to give C++ the ability to speak the languge of **R**: vectors, matrices, data frames, time series, etc. with the goal of faciliting communication between the two languages, and more importantly, improving the performance of **R** packages.

When an **R** function calls a C++ function using the `.Call` interface the C++ side sees each parameter as a pointer to the corresponding **R** object. The C++ class that models each **R** type has a constructor that takes this pointer as an argument; it uses it to construct the model object in the address space of the C++ module. The C++ classes also have constructors that take C++ data structures (instead of a pointer to an **R** object), and objects constructed using them can be passed to **R** just as easily as those constructed from **R** pointers.

*This is an updated version of the document *Building* **R** *Packages that call C++ functions:* **Rcpp***: R/C++ Interface Classes Version 5.0*, September 20, 2006. I am grateful for helpful comments from Dirk Eddelbuettel, Hin-Tak Leung, Uwe Ligges, Duncan Murdoch, Brian Ripley, and Paul Roebuck.

To return values to **R** or to make calls to **R** functions the process is reversed: each C++ object needs to be transformed into an object in **R**'s address space, and a pointer to this object must be passed to **R**. A pointer to an **R** object as seen on the C++ side is called a SEXP,[1] and the protocol just described can be made more explicit using the code fragment:

```
RcppExport SEXP CppFunction(SEXP sexp, SEXP fsexp) { // C++ declaration.
    RcppFunction f(fsexp,1); // Construct C++ interface f from fsexp (numArgs=1).
    RcppType1 c1(sexp); // C++ constructs object c1 of type RcppType1 from sexp
    ··· // Do some processing, generate C++ object c2 to be passed to R
    RcppSEXP sp = getSEXP(c2); // Map C++ object c2 (type RcppType2) to SEXP sp.
    f.setNextArg(sp); // Set first and only argument for function call.
    SEXP result = f.callR(); // Call R function.
    RcppType3 c3(result); // Instantiate return value as C++ object.
```

Here the function `CppFunction` is called from **R** using the `.Call` interface with two parameters, `sexp` and `fsexp`. We assume that the first points to an **R** object that can be modeled using `RcppType1`, and the second points to an **R** function that can be modeled using `RcppFunction`. We assume that the function accepts 1 argument of the **R** type mapped to `RcppType2`, and returns a value that can be mapped to `RcppType3`. The pattern should be clear from the code fragment: C++ constructors are used to transform **R** objects into C++ objects, and the overloaded function `getSEXP()` is used to do the reverse transformation.

The function `getSEXP()` transforms its input C++ object into the corresponding object in **R**'s address space and returns a pointer (or SEXP) to this **R** object. It also returns a count of the number of addresses in **R**'s address space that had to be protected from cleanup by **R**'s garbage collection. The **Rcpp** library takes care of unprotecting these addresses before returning to **R** so that the corresponding memory can be freed when no longer in use.

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 (both Date and POSIXt types), factors, data frames, zoo time series objects, and function objects. A summary of the main classes is given in Table 1.

Description	**R class**	**Rcpp class**
Parameters	`list`	`RcppParams`
Date	`Date`	`RcppDate`
Date-Time	`POSIXt POSIXct`	`RcppDateTime`
Date details	NA	`RcppLocalTime`
Factor	`factor`	`RcppFactor`
Data frame	`data.frame`	`RcppFrame`
Zoo time series	`[zooreg] zoo`	`RcppZoo`
Frame column	NA	`RcppColumn`
Function adapter	`function`	`RcppFunction`
Return list	user-defined	`RcppResultSet`

Table 1: Mapping between selected **Rcpp** classes and **R** classes.

Only regular fixed-frequency zoo objects have the `zooreg` class attribute. Note that a `ts` (fixed-frequency) time series can be transformed into an equivalent `zoo` time series using `as.zoo()`, and back again using `as.ts()`. Thus fixed-frequency time series can be handled by encapsulating them in zoo

[1]The term SEXP comes from the Lisp community where it stands for Symbolic EXPression. Unlike Lisp everything is not a list (or dotted pair) in **R**; vectors are used for faster computations. On the other hand, like Lisp a list structure is used to represent function expressions.

objects. Similarly, a `tseries` irregularly spaced time series (class `irts`) can be transformed into a `zoo` time series and back again using `as.zoo()` and `as.irts()`, so this type of time series object can be handled as well. The extensible time series class `xts` can be transformed and managed similarly.

There are two primary use cases. In the first **R** calls C++ functions that do most of the heavy computation at the speed of compiled code, and **R** is primarily used to view and perform statistical analysis on the results. In the second C++ code makes calls to **R** functions that are easily modified in an interactive fashion. In the first case the goal is faster computations and in the second it is flexibility.

Note that when a call to a C++ function returns to **R** all objects created during the call are destroyed. Thus if persistence is important it will have to be implemented outside of the C++ object system.

A straight-forward solution is to simply use **R** to maintain state and to recreate C++ objects as needed during the calls. To limit the amount of copying involved in making repeated calls to C++ functions it is possible to work directly with **R**'s address space, and there are classes that help you to do this, but they are intended to be used in situations where one has to work with huge data sets where the copy overhead is not acceptable. These facilities should not be used routinely because this is inconsistent with the pass-by-value semantics of **R**.

The **Rcpp** library obviously uses internal information about **R** and for this reason there is the danger that changes made to **R** will break **Rcpp**. Mitigating this risk is the fact that the **R** types modeled are well-established and are unlikely to change in ways that would break existing **R** scripts. Another mitigating factor is that the underlying Lisp-like structure of **R** is simple and unlikely to change (vectors, lists, attributes, etc.). In the event that a class like `zoo` time series is changed by the addition of new attributes, say, it would be relatively easy to update the corresponding C++ constructor and `getSEXP()` function to accommodate this change.[2]

The remainder of this paper is basically a User's Guide for the **Rcpp** library. We explain in detail how to build and test **R** packages that employ **Rcpp** under UNIX and Windows. To this end we have created an **R** package named `RcppTemplate` that can be used as a template when creating your own packages. It includes a test driver (`RcppExample.cpp`) that illustrates how to use all of the **Rcpp** classes. The process of linking to external C++ libraries like `QuantLib` is also explained.

This document and the *Rcpp Quick Reference* can be displayed after loading the `RcppTemplate` package by issuing the **R** commands `showRcppDoc()` and `showRcppQuickRef()`, respectively.

2 The `RcppTemplate` Package

The `RcppTemplate` package was designed to be used as a template for creating packages that make use of C++ code or external libraries with the help of the **Rcpp** interface class library. Basically to create a new package starting from the template package replace the name "RcppTemplate" everywhere it occurs with your package name, then modify the source, data, demo, and documentation files as needed.[3]

To get started let's download and run the template package sample code. We will use it to illustrate how **Rcpp** is used. The `RcppTemplate` package is available at the official **R** web site, `http://cran.r-project.org`. By navigating to the packages archive and going to the `RcppTemplate` package page it an be checked that all packages have a source file named `<PackageName>_<version>.tar.gz`, and most have a Windows binary with the same name except that "`.tar.gz`" is replaced with "`.zip`," and a MacOS X version where the suffix is "`.tgz`". To submit your own package to the CRAN archive only the source archive needs to be sent. This will be unpacked, checked, compiled, and installed, and all relevant information about the package will appear on the package page.

The official reference on writing **R** extensions is *Writing R Extensions*, available at the CRAN site. This document should be consulted for additional information about the C API that is the foundation for **Rcpp**. It should be downloaded along with the source archive for `RcppTemplate` for later reference.

[2] Given an **R** object `x`, use `str(x)` to look at its structure, and use `attributes(x)` to look at its attributes. This is the information that drives the C++ constructor and is set by `getSEXP()`.

[3] Remove the file `inst/doc/RcppDoc.Rnw` and its `Makefile` before starting work on your own package as this so-called vignette file triggers extra processing at build time.

After installing **R** the `RcppTemplate` package can be installed directly from the CRAN archive by using the **R** command:

```
> install.packages("RcppTemplate")
```

You will be presented with a list of host names from which the package can be downloaded. Select one that is reasonably close. If you do not have permission to write to the directory where **R** was installed you may be asked for permission to install the package into your personal file area (say yes, or change the permissions on the **R** directory).

After installing the package it can be loaded using the `library` command, and the example function `RcppExample` can be run as follows:

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

Actually, the function `RcppExample` takes a long list of parameters and it would be messy to make an explicit call here. Instead we run the **R** code that appears as an example in the manual page for `RcppExample`. This **R** code can be viewed by looking at the man page as follows:

```
> ?RcppExample
```

Scroll down to the "Examples:" section where you will find the actual call; it looks like:

```
result <- RcppExample(params, nlist, numvec, ...
```

`RcppExample` returns a list of objects and assigns this list to `result`.

Running the example code defines the variable `result`, and if you type this variable name on a line by itself a list of names appears. These are the names of objects on the return list. To view the object corresponding to a particular name, say `zoo8`, type:

```
> result$zoo8
```

The reason why just the object names are printed and not the object contents (when `result` was entered) will be explained shortly. The object `result$zoo8` is of class `zoo`, and if this expression is typed on a line by itself the contents of this object are displayed.

Let's scroll back up and take a look at the input parameters on the man page example. The `params` parameter is just a list of name-value pairs, where the values can be strings, integers, or dates (if the C++ code attempts to fetch a value that does not have the correct type an exception is thrown, and you bounce back to **R**). The `nlist` parameter is very similar, and indeed this particular type is redundant and should probably be removed. The nature of the numeric and string vector parameters should be obvious.

To assign three consecutive days to `datetimevec` we start with the current time and add multiples of $60 * 60 * 24$, because **R** measures time in seconds for this type (`POSIXt`). On the other hand, to do the same thing for the `Date` vector `datevec` this scaling is not necessary because **R** measures time in days for this type. On the C++ side both date types measure time in the same units, and mixed comparisons are allowed, with caveats.

The definitions of the factor `myfactor` and the data frame `df` should be clear (I assume the reader has basic familiarity with **R**).

The `zoots` parameter is set equal to an irregularly spaced time series indexed by Dates.

The `func` parameter is an **R** function that simply checks the type of its arguments and returns a vector, and the `hypot` function computes the distance to the origin. We will show how to call these functions from C++ later.

3 Fetching R Data From the C++ side

So what does the C++ side of this call look like? Something like this:

```
RcppExport Rcpp_Example(SEXP params, SEXP nlist, SEXP numvec, SEXP numat,
        SEXP df, SEXP datevec, SEXP datetimevec, SEXP stringvec,
        SEXP fnvec, SEXP fnlist, SEXP zoots, SEXP myfactor) {
        /** Skipping and paraphrasing--see RcppExample.cpp for the truth.  */
        RcppParams rparms(params);
        RcppNumList nl(nlist);
        RcppVector<double> vecD(numvec);
        RcppMatrix<double> matD(nummat);
        RcppFrame frame(df);
        RcppFactor factor(myfactor);
        RcppDate dateVec(datevec);
        RcppZoo ts(zoots);
        /** Omitting lots of stuff.  */
```

All of the input parameters appear as SEXP's, and the corresponding C++ objects can be constructed easily from these SEXP's as shown here.

We now have a number of C++ objects that model the input **R** data structures. These objects can be used like this:

```
        double tolerance = rparam.getDoubleValue("tolerance");
        RcppDate startDate = rparam.getDateValue("startDate");
        double x = matD(i,j);
        string observationStr = factor.getObservationLevelStr(i);
        int observationNum = factor.getObservationLevelNum(i);
        int numObservations = factor.size();
        vector<RcppColumn> cols = frame.getColumns();
        RcppDateTime dt = cols[6].getDateTimeValue(row);
        double ExcelVal = dt.numExcelPC();
        vector<vector<double> > zoodata = ts.getDataMat();
        vector<RcppDate> zooindex = ts.getIndDate();
        vector<int> perm = ts.getSortPermutation();
        // ordered rows:  zooindex[perm[i]], zoodata[perm[i]][j]
```

The first few examples should be clear. In the case of `RcppFactor` factor levels are represented by strings or by levels numbers, is in **R**, and the size of a factor is the total number of observations. Data frames are basically a vector of columns (of type `RcppColumn`), and data is fetched naturally by column, then by row, as in the case of the date-time variable `dt`.

Zoo objects can be indexed vectors or matrices (with indexed rows), and the permutation vector needed to sort the data by the index is maintained for the case where the user creates the zoo object on the C++ side (the data is already sorted when received from the **R** side so the permutation is the identity in this case).

The value ExcelVal, if placed into an Excel spreadsheet on a PC and formatted as a date, would display as the date and time corresponding to `dt`. This would not work under MacOS (at least not by default). But this problem is solved by using `dt.numExcelMac()` instead. This returns the default value used by Excel on a Mac. There is also `dt.numJulian()` and `dt.getRValue()`. The first function returns the julian day number (fractional day part dropped), and the second function returns the value used by **R** to represent this date object. All of these functions work for both `RcppDate` and `RcppDateTime` (the latter is a subclass of the former). For convenience these functions are also implemented at the **R**

script level and included in the package namespace—see the man page for `DateNum` when `RcppTemplate` is loaded.

Incidentally there are a few convenience functions that do things like return the weekday corresponding to a particular date (values `RcppDate::Mon`, `RcppDate::Tue`, etc.), that find the next occurrence of a particular weekday, or that find the n-th occurrence in a given month.

There is also a useful template function `to_string(obj)` that returns the string value of any object that knows how to stream itself.

A Quick Reference on all of the **Rcpp** classes can be found in `inst/doc/QuickReference.txt`. It is basically a stripped down version of `Rcpp.hpp` formatted to show all public methods and the relationship between the classes. For more detail look at the comments in `Rcpp.hpp` and `Rcpp.cpp`. Locating these source files is the next topic.

4 Package Structure and Source Code

As mentioned above the source archive (the `.tar.gz` file) can be downloaded from the CRAN site. It can be extracted into the current directory using:

```
$ tar -xvzf RcppTemplate_<version>.tar.gz
```

Under Windows a `tar` command is available as part of the Rtools kit (see Section 7).

The root of the package directory tree will be named `RcppTemplate`, and the main components are `src`: where your C++ source code should go, `man`: man page directory, `R`: R scripts that actually expose your C++ functions, and `RcppSrc`: contains the **Rcpp** source code and should not be modified by you (unless you find problems). There are also (optional) `demo` and `inst` directories. The first can contain demo scripts that can be run using the **R** `demo()` command, and the second directory can contain documentation, data, and license files that you want to be part of the installed package (note that the source file directories are NOT part of the installed package or the Windows `.zip` file).

Other necessary parts of an **R** package are located in the root directory (named `RcppTemplate`). These include the `DESCRIPTION` file and the `NAMESPACE` file, and for improved portability, a `configure` script. The `DESCRIPTION` file gives a brief description of the package, version info, author, dependencies, etc., and the `NAMESPACE` file lists the functions that you want exposed by this package (and not confused with functions of the same name in other packages).

The `NAMESPACE` file shows the name of the dynamic library to be load when this package is loaded (using the `library` command), and it also shows the exported **R** function names. These functions are usually defined by scripts in the `R` subdirectory that are run when the package is loaded.[4]

Recall that the function called in the example page was named `RcppExample`, not `Rcpp_Example`. The first symbol is defined in the **R** script `R/RcppExample.R`, and the second symbol is defined in the C++ code `src/RcppExample.cpp` and made available via the shared library `RcppTemplate.so` (or `.dll`, as the case may be). The **R** script is basically a wrapper around the call to the C++ function, which looks like this (see `R/RcppExample.R`):

```
result <- .Call("Rcpp_Example", params, nlist, numvec, ...,
PACKAGE="RcppTemplate")
class(result) <- "RcppExample"
```

Here `result` is assigned the list of objects returned by the C++ code in `Rcpp_Example`, and then this list is given the class attribute `RcppExample`. The purpose of the class attribute is to override the default behavior of functions like `print()` that are automatically called when the name of a variable is used

[4]Use the `search()` command to view the package search list (and order), and use `ls(package:RcppTemplate)` to view the objects in this package including namespace exports. Use `str(obj)` to view the structure of a particular object.

in certain contexts. The default behavior of `print()` is to print everything recursively, and this would generate too much output for a large object.

The work-around is to define `print.RcppExample` in the script file (and to export this name in the `NAMESPACE` file). The result is that the command `print(obj)` when `obj` has class `RcppExample` results in the call `print.RcppExample(obj)`, a specialized function call for objects of this class. In this case it simply prints the names of all of the objects on the list returned by `Rcpp_Example`, and not the contents of those objects. This is what happens when the example is run: if after running the example you type `result` on a line by itself this is equivalent to `print(result)`, which is dispatched to `print.RcppExample`, and the result is that a list of object names is displayed, but not the object contents.

As another example, consider a regular (fixed-frequency) zoo object. It has class `c('zooreg', 'zoo')`, a vector of strings in this case, where the first string represents the "derived part" and the second is the "base part." The command `print(obj)` for an object of this type will get dispatched to `print.zooreg(obj)` if this function is defined, otherwise it will be dispatched to `print.zoo(obj)` if defined, otherwise it will be dispatched to `print.default(obj)`. Of course, there is nothing special about `print` here, and the same class-based dispatch mechanism works for any functions that are made available via the `NAMESPACE` file.

Finally, let's look at the source code for the **Rcpp** sample client application `src/RcppExample.cpp`. The main function there is named `Rcpp_Example`, and you will note that there is a qualifier `RcppExport` in front of the definition. This ensures that this function name is available when the package shared library is loaded. The pattern that should be followed by C++ modules that are called by **R** (and that use **Rcpp**) is shown in Figure 1.

```
#include "Rcpp.hpp"
using namespace std;
using namespace Rcpp;
RcppExport SEXP RcppSample(SEXP params, SEXP a) {
    SEXP rl=R_NilValue; // Return this when there is nothing to return.
    char* exceptionMesg=NULL;
    try {
        /** Create objects from SEXP's and do some work.  */
        RcppResultSet rs; // result set
        rs.add("name1", result1); // add items to result set
        rs.add("name2", result2);
        ...
        rl = rs.getResultList(); // done, get return list.
    } 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**.

The `using namespace` lines are optional. If you want to reduce namespace clutter you can remove them and be more explicit: `std::vector`, `Rcpp::RcppResultSet`, etc. We have already discussed `RcppExport`, SEXP's, and how to create C++ objects from SEXP's. After all of the object creation and calculations are done the results are returned as a list of objects with the help of the `RcppResultSet` class (look at `RcppExample.cpp`).

Objects are added to the return list by overloaded `add()` methods (see the definition of `RcppResultSet` in `Rcpp.hpp`). These methods call `getSEXP()` to get the corresponding SEXP pointer, and then add this pointer to the list with the specified name attached.

The C++ try/catch mechanism works well in spite of the fact that **R** is not written in C++. The **Rcpp** library does a fair amount of type checking, bounds checking, and other sanity checking and throws an exception if there are problems. The message displayed by the catch clause can then be searched for in the **Rcpp** source files to help identify the problem.

The sample file `src/RcppExample.cpp` illustrates how to use most of the **Rcpp** classes, and it actually does something non-trivial with the input data: it computes a schedule of dates between two given dates that fall on the n-th occurrence of a specified weekday.

The date conventions illustrated and explained in `RcppExample.cpp` are as follows

1. Dates can be instantiated from **R** SEXP's as explained above, or from the C++ side by giving month/day/year, and in the case of RcppDateTime, a fraction of day component, so .75 would translate into 18:00 GMT, for example.

2. Subtracting two dates yields the number of days between them, and this applies to both date types, and even to a mixture of the two types (must be interpreted with care due to time zones). The result can include a fractional part down to a fraction of a second (in the case of `RcppDateTime`).

3. Increments to dates are in units of days, so `date++` increases `date` by 1 day, and this applies to both date types.

4. To add n seconds to an `RcppDateTime` object, add n/`RcppDate::DAYS2SECS` days.

5. To inspect the underlying m/d/y or other information about an `RcppDate` or `RcppDateTime`, use the `RcppLocalTime` class.

6. Both date types can be streamed in the usual way.

5 Calling R Functions from C++

There is a demo function in `src/RcppExample.cpp` that illustrates how to call **R** functions from C++. The pattern is as follows. Assume that you have written an **R** function named `vol` that accepts two real parameters, `T` and `K`, and returns a real-valued result. To call this function from C++ we must pass `vol` as a parameter to the `.Call` function. On the C++ side we must know how many parameters the function takes, as well as the types expected as parameters and return values.[5]

If the input SEXP is also named `vol`, then the `RcppFunction` object is created and used to call the **R** function like this:[6]

```
RcppFunction f(vol,2);
f.setNextArg(getSEXP(T));
f.setNextArg(getSEXP(K));
```

[5]Like Lisp and Smalltalk, **R** objects are dynamically typed, so types are not as easily determined as they are on the C++ side. It is important that **R** functions that are called from C++ check their input parameters and if the types are invalid use the `stop()` function to issue an error message and terminate.

[6]We use `Rprintf` instead of the `iostream` library for debugging purposes because `iostream` does not work on some systems (Windows), but this comes at a cost: `Rprintf` is not type-safe and if you pass it a `std::string` where it expects a `char*` **R** will crash—use `c_str()`.

```
SEXP result = f.callR();
RcppVector<double> vec(result);
Rprintf("Return value = %lf\n", vec(0));
```

Here the second parameter to the constructor is the number of arguments expected. The use of an `RcppVector` to capture a single real value is not natural, but it works. Note that after every `callR` the arguments are reset so they must be set again before the next `callR`. Data frames, time series, and other supported types can be passed to (and returned from) **R** functions in the same way.

6 Building and Testing a Package Under UNIX

Under UNIX the `RcppTemplate` source package can be built and installed into a local directory named `Library.test` as follows. Unpack the source archive, change directory to the parent of `RcppTemplate`, and run:

```
$ mkdir Library.test
$ R CMD INSTALL -l Library.test RcppTemplate
```

The directory `Library.test` will contain `RcppTemplate`, the root of the installed package, and under this will appear a `libs` subdirectory containing the shared library for `RcppTemplate`.[7]

If you leave out the `-l Library.test` part, the package is installed into the standard place, which is what happens when you use `install.packages()`. An important advantage of using `install.packages()` is that it can determine what packages this one depends on and download and install those packages automatically.

To run the package from this test location start **R** and type:

```
> library(RcppTemplate, lib.loc="Library.test")
```

Use a full path name if you are not immediately above `Library.test`. Even better, place the library command in a function that is defined in your `.Rprofile` start-up file:

```
dotemplate <- function() {
    library(RcppTemplate, lib.loc="/usr/home/work/Library.test")
    example(RcppExample)
}
```

Change the pathname as needed. This saves a lot of typing and is useful while you are going through the testing phase.

When you are satisfied that the package functions properly the package structure can be checked using:

```
$ R CMD check RcppTemplate
```

This does a thorough consistency test and should be passed before submitting your package to CRAN. If everything goes well the final source archive can be created for submission using:

[7] Because the **Rcpp** library makes use of C++ templates it sometimes happens that there are undefined references when you build the pacakge in this way and there are object and library files from a previous build. To resolve the problem delete the binary files and build starting from a "clean" state. Under UNIX the `cleanup` script can be used for this purpose; under Windows use `sh cleanup.win`.

```
$ R CMD build RcppTemplate
```

This will create `RcppTemplate_<version>.tar.gz`, where the version is taken from the `DESCRIPTION` file.

While it is possible to use a traditional `Makefile` to resolve dependencies and build the package, it is simpler to use a `Makevars` file. The purpose of this file is to define `PKG_CPPFLAGS` and `PKG_LIBS`, and to build the **Rcpp** library. The `Makevars` file (in the `src` subdirectory) that is automatically generated by the `configure` script in the `RcppTemplate` package (based on `Makevars.in`) follows:

```
PKG_CPPFLAGS = -I../RcppSrc
PKG_LIBS = -L../RcppSrc -lRcpp
RcppLib = ../RcppSrc/libRcpp.a
RcppSrc = ../RcppSrc/Rcpp.cpp ../RcppSrc/Rcpp.hpp
.PHONY: all
all:  $(SHLIB)
$(SHLIB): $(RcppLib)
$(RcppLib):  $(RcppSrc)
        (cd ../RcppSrc; make)
```

The only part that may require changes based on the needs of your package are the first two lines, and the primary purpose of the `configure` script is to make the necessary changes based on the system configuration where it is run.

If you know what the compiler flags and library locations should be for your system you can simply modify the first two lines appropriately and rename the `configure` script to `configure.save`, say, so that `Makevars` is not overwritten.

Before submitting the package to CRAN the `configure` script will need to be modified so that any necessary edits are made automatically and transparently to the package installer.[8] This does not work under Windows, so instead there is a custom `Makevars.win` file—see next section.

For example, the directory `inst/doc/QuantLib` contains sample `Makevars.in` and `Makevars.win` files that can be used to link against the `QuantLib` C++ quantitative finance library. It also contains an example distributed with that library. The only changes made to the example were to replace the `main()` declaration with an **Rcpp**-styple entry point, and to replace all return values with `R_NilValue`. See the `README` file in that directory for more information. The `RQuantLib` package illustrates how this can be automated and extended.

7 Building and Testing a Package Under Windows

The procedure for building and testing **R** packages under Windows is very similar to the one under UNIX, but this requires installing the following tools:

- The UNIX tools for **R** (Rtools) from http://www.murdoch-sutherland.com/Rtools,
- The MinGW GNU compiler,
- MikTeX (TeX for Windows),
- Microsoft's HTML Help Workshop (used to make chm files)

[8] This process only applies to UNIX and is obscure. Run `autoheader` in the package root directory to generate `src/config.hpp.in` from `configure.in`. Then run `autoconf` to generate the `configure` script and `src/config.hpp`. The package build process will start by running the `configure` script, and this will create `src/Makevars` by filling in the symbolic values that appear in `src/Makevars.in`. The entire process is driven by the `configure.in` file, and this is not difficult to follow. See the GNU docs on `autoconf` for more information.

The last two items are not essential for development purposes, but without them package documentation will not be generated in certain formats. The first two items can be installed together because the Rtools kit installation gives you the option of installing MinGW.

To be sure that all necessary programs are found (and to easily update the process when you install new versions) you could run a script like this one (`setrpaths.bat`):

```
set PATH=c:\Program Files\R\R-2.9.2;%PATH%
set PATH=c:\Rtools\bin;%PATH%
set PATH=c:\Rtools\MinGW\bin;%PATH%
set PATH=c:\Rtools\perl\bin;%PATH%
set PATH=c:\Program Files\MikTeX 2.7\miktex\bin;%PATH%
set PATH=c:\Program Files\HTML Help Workshop;%PATH%
```

Of course, the names will have to be modified to match your system configuration and installed versions. This would typically be run before each development session.

With the tools installed and the path variable set we can now install a test version of `RcppTemplate` in `Library.test` just as we did in the UNIX case (see last section). This should not be done in directories that have names with spaces in them like "`My Documents`" as this will probably cause the build to fail. It should also be kept in mind that Windows uses backward slashes while UNIX uses forward slashes in file name paths (the Rtools binaries accept slashes in either direction, but Windows tools tend to be less accommodating).

The package check and package build process is also exactly the same as in the UNIX case. There is no `configure` script for Windows, and instead an OS-specific `Makevars.win` file is used. Simply modify the variables `PKG_CPPFLAGS` and `PKG_LIBS` as explained in the last section based on the needs of your package.

Finally, a Windows binary (`.zip` file) can be created using:

```
$ R CMD --no-vignettes --force --binary --use-zip RcppTemplate
```

Note that it is not necessary to submit a Windows binary version to CRAN. The Windows binary will be generated by CRAN from the submitted source archive (`.tar.gz` file).

Under Windows a package can be installed from a local zip file (instead of using `install.packages()` to fetch it over the network) by using the menu option: Packages / Install from local zip files.

8 Package Demo and Data Files

Package demo scripts can be inserted into the `demo` subdirectory. These will typically be larger scripts that do not fit naturally into man pages. To display the list of demo scripts available and then run one use commands like:

```
demo(package="RcppTemplate")
demo(SincSurface)
```

The package has to be loaded for this to work. The demo descriptions displayed by the first command must be inserted into `demo/00Index`.

The demo script `Peacock.R` illustrates two kinds of dependency that can occur. It depends on another package named `ReadImages` that must be installed (from CRAN), and this package in turn depends on a jpeg shared library that must be installed at the OS level.[9]

[9] Under Windows you may need to add the shared library location to your search path (or place it in a system directory) so that it will be found at run-time. Under UNIX you may need to use the `ldconfig` command to accomplish the same objective.

Package data files, images, license files, and other information that you want to be part of the installed package should go into the `inst` subdirectory. The `RcppTemplate` source package contains `inst/datasets/exam.txt`. When the package is installed the root directory will contain `datasets/exam.txt`. To access the file `datasets/exam.txt` in your scripts use code like:

```
myfile <- system.file('datasets', 'exam.txt', package='RcppTemplate')
my.frame <- read.table(myfile, header=TRUE)
```

For more information see the help page for `read.table()`, for example.

9 R Package Preparation Checklist

To conclude, here is a summary of the steps necessary to create an **R** package:

- Place your C++ (or C or FORTRAN) source code into the `src` subdirectory. (Skip this and related steps if your package contains only **R** scripts, no source code to be compiled.)
- Set the compiler flags in `Makevars` and `Makevars.win` so that any needed external include or library directories will be found at build time. For improved portability (on UNIX platforms) use the template `Makevars.in` together with a `configure` script.
- Update the information in `DESCRIPTION` to show package description, version, date, author, dependencies, etc. It is important to include dependencies so that `install.packages()` knows what is required. A long description can span several lines provided all lines after the first begin with a space or a tab.
- Insert **R** scripts that define all of your package functions into the `R` subdirectory, and optionally add demo scripts to the `demo` subdirectory and update `demo/00Index`.
- Update the `NAMESPACE` file to show the package shared library name and all exported function names. A start-up message can be issued when the library and namespace are initialized by defining a function named `.onLoad`.
- Insert specially formatted help pages (with `.Rd` suffix) into the `man` subdirectory. For example, the `RcppExample` man page is `man/RcppExample.Rd`. As was done in this case, it is good practice to include an examples section on each man page.
- Make sure the package and documentation are consistently defined by running the CRAN check function, and when all is well use the build command to generate the source archive for distribution.

The `inst` directory contains `LICENSE-Rcpp.txt` and `LICENSE.txt`. Both files will be copied to the root directory of the installed package. If desired license information can be inserted into the latter file (license information must appear in the `DESCRIPTION` file). Refer to the "Writing R Extensions" manual at the CRAN site for additional information on package creation, man page formatting, and the **R** API.