rcppbind: A **R**/C++ template class library

Gang Liang <gumpleon@gmail.com>

July 1, 2008

Abstract

Being one of the most powerful open-source statistical softwares, **R** is flexible and easy to master for statisticians. But unfortunately, it is quite difficult for developers to write softwares for **R** using some low-level programming languages such as C++: under such circumstances, developers are forced to learn a great deal of internal structures and low-level operations of **R** in order to manipulate **R** objects. The goal of this template library, `rcppbind`, is to provide a simple interface between C++ and **R**, so **R**/C++ developers can access **R** objects in usual C++ ways. It is the wish of the author that the burden on developers could be lessen in some degree.

1 Introduction

The package `rcppbind` installs necessary C++ header files and help files, and provides several demos on how to use the template library. It contains only two **R** functions:

1. `rcppbind.demo`: some demonstration modules using the rcppbind template class library;
2. `rcppbind.path`: return the path to the installed header files.

1.1 Library Design

Different from C++ being a strong-type language, **R** is of weak type. In other words, all **R** objects are handled through an *SEXP* (simple expression) interface, which is an opaque pointer. Each **R** object has a class type, such as `vector, matrix, list, data.frame`, and a data type, such as `int, double, Rcomplex, char*, Date, POSIXct`. Some examples of **R** objects are a vector of integers or a matrix of doubles. A simple but important fact is that even a single integer in **R** is indeed a vector of integer of length 1. R

The basic idea of `rcppbind` is to define a set of classes to wrap **R** objects in order to minimize the effort of **R**/C++ developers in mastering many low-level **R** operations. Roughly speaking (not precise), class types can be viewed as wrappers for data types. Currently, the following four class types are supported by the `rcppbind` template library:

R	`vector`	`matrix`	`data.frame`	`list`
`rcppbind`	`rvector`	`rmatrix`	`rdataframe`	`rlist`

Among them, `rvector` and `rmatrix` are template classes, while `rdataframe` and `rlist` are regular classes. These classes serve as a wrapper for fundamental data types. Details on how to wrap **R** objects can be found in Sec 2 and Sec 4.

Not all data types are born equal: some special data types need special treatments, most noticeably `char*`. The following `rcppbind` classes are designed to wrap these outliers:

R	char*	Date	POSIXct
rcppbind	rstring	rdate	rtime

A detailed account of these special data types can be found in Sec 3.

In order to simplify the library design, the following design specifications are imposed:

1. The library only supports `.Call` interface in **R**. This decision is made simply for convenience. Interested readers please refer to **R** manual "Writing **R** Extensions".

   As a consequence, the `rcppbind` library only supports the positional parameter-matching calling convention, which is natural for C++. Since **R** also supports a name-matching calling convention: it can be implemented inside the **R** function before the `.Call` interface.

2. In order to bind with `rcppbind` objects, one needs to know both class and data types of **R** objects: both class and data types have to be matched.

   An exception to this rule is that internal conversions will be performed for certain commonly convertible data types. For instance, a vector of integer in **R** will be automatically converted to `rvector<double>` if such type is declared to wrap the input **R** object. An exception will be thrown If the data type does not match and a conversion is not possible.

3. In **R** an object has to be protected before accessing its data area. This protect/unprotect mechanism is automatically enforced under the template library framework: a `rcppbind` object is protected when constructed, and unprotected when deleted.

   In order to be 100% worry-free, the template library uses `UNPROTECT_PTR` to unprotect objects. The normal use of `PROTECT` will cause some potential problems since it is not the library but the developer who dictates the order of the object creation and deletion.

   The use of `UNPROTECT_PTR` over `PROTECT` has a small performance penalty. Hence, it is strongly recommended (but not required) that library users declare objects in a first-declare-last-delete fashion.

1.2 Library Usage

It is author's presumption that such a template library is used to code some computational intensive modules by taking advantage of efficiency of C++ over **R**, and easiness of C++ over C. The general flow of a **R**/C++ module would be

1. wrap **R** input parameters;
2. do operations using whatever C++ data structures, functions, and tricks;
3. return computation results to **R**.

There are two important questions from a developer's point of view:

1. how to enter and exit a `rcppbind` module, what syntax?
2. how to bind **R** objects inside a `rcppbind` module?

These questions are of course interrelated. Below we will first show how **R** objects are binded in `rccpbind`: it contains more details. On the other hand, it might be a good idea to read Sec 4 first on the entrance and exit of a `rcppbind` module to get an overview sense of the template package.

2 Class Types

In the current implementation, the template library only supports four class types. Internally, `vector, list` are the two most fundamental class types: all other class types are derived from them. For instance, `matrix` is internally `vector`, `data.frame` is implemented through `list`.

2.1 `rvector`

The class template `rvector<T>` is designed to wrap **R** vector objects. The currently supported data types, `T`, are `int, double, Rcomplex, char*, rdate, rtime`. Some of the most useful member functions of `rvector<T>` are

1. `rvector<T> (SEXP)`: constructor to wrap an existing `vector` object
2. `rvector<T> (int)`: construct a new `rvector<T>`
3. `int length()`: return the length of the vector
4. `T* begin(), end()`: return the begin and end of the iterators of the vector
5. `T& operator [int], T& operator [std::string]`: subset operator
6. `operator =, *=, +=, -=`: (for numeric data types only)

Notes

1. It is preferred to use iterators returned by `begin(), end()` to access members of a **R** vector object. There is a small penalty of using `operator []`: the validity of the index will be checked when when each time called.
2. The subset `operator []` is overloaded: it also supports searching elment by dimname – a `std::string` object.
3. `rvector<rstring>, rvector<rdate>, rvector<rtime>` are wrappers for special data types. See Sec 3 for more details.

2.2 `rmatrix`

The class template `rmatrix<T>` is designed to wrap **R** matrix objects. The currently supported data types `T` are `int, double, Rcomplex, char*, rdate, rtime`. A list of important member functions of `rmatrix<T>` is as follows:

1. `rmatrix<T> (SEXP)`: constructor for an existing **R** matrix
2. `rmatrix<T> (int nrow, int ncol)`: constructor
3. `int nrow(), ncol()`: return the dimension info of the object
4. `T* begin(), end()`: the begin and end iterator of the whole data block
5. `T* cbegin(int n), cend(int n)`: the begin and end iterators of the nth column;
6. `T* cbegin(std::string), cend(std::string)`: the begin and end iterators of the column with the given name;

7. `T& operator (int nrow, int ncol)`: element access
8. `operator =, *=, +=, -=` (for numeric data types only)

Notes

1. Similar to `rvector`, it is preferred to use iterators returned by `cbegin, cend` to access column members of a **R** matrix object: **R** is a column-prior language. The element access operator `()` would check the validity of the index each time when called.
2. `rmatrix<rstring>, rmatrix<rdate>, rmatrix<rtime>` are wrappers for matrix of special data types. See Sec 3 for more details.

2.3 rdataframe

The class `rdataframe` is designed to wrap **R** `data.frame` objects, which are data tables of rectangular shape. `rdataframe` is a regular class, and we need to further bind each column of a `rdataframe` object with a `rvector` object to retrieve data from a data.frame object. A list of important member functions of `rdataframe` is as follows:

1. `rdataframe(SEXP)`: constructor for wrapping an existing SEXP
2. `rdataframe(int, int, std::vector<const std::type_info*>&, std::vector<std::string>&)`: create a new data.frame object
3. `rvector<T> getColumn(n)`: retrieve the nth column of the data.frame
4. `rvector<T> getColumn(colname)`: retrieve the column with the given column name
5. `SEXPTYPE getColType(n)`: return type of the nth column
6. `int nrow(), ncol()`: return the dimension information
7. `std::vector<std::string>& getColNames()`: return the column names;
8. `void setColNames(std::vector<std::string>&)`: set column names

Notes

1. Again, one needs to know both class and data types of a column in order to bind it with a `rvector` object. The usual syntax for applying the template member function `getColumn` is as follows:

   ```
   rdataframe df = ...
   rvector<double> datavec = df.getColumn<double>(0);
   ```

2. In order to construct a `rdataframe` object of interest, one needs to provide the dimension information, each column type info, plus column names. The column type info is denoted as `std::vector<const std::type_info*`. The reason to use type info instead of `SEXPTYPE` is because it contains the complete information to determine the class type. See `demo_df.cpp` for an example.

2.4 rlist

The class `rlist` is designed to wrap **R** `list` objects: `list` is the most flexible class type in **R**. A list of important member functions of `rlist` is as follows:

1. `rlist( SEXP sexp )`
2. `rlist( int size, std::vector<std::string> strname = NULL )`
3. `int size()`: return the size of the list
4. `void setElement( int index, SEXP sexp )`
5. `SEXP getElement( int index )`

Notes

1. The above `getColumn` and `setColumn` member functions can also take a `std::string` argument for searching elements by name.
2. Both functions `getColumn` and `setColumn` takes or returns parameter of type `SEXP` because the underlying objects are unclear to the class itself. Nevertheless, the `rcppbind` framework provides a general approach for handling these opaque `SEXP` pointers. Generally speaking, module writers know the types of these objects; hence, it is his/her responsibility to manually wrap these objects with the corresponding `rcppbind` classes. See `demo_list.cpp` for a simple example.

3 Data Types

We discuss three special data types in this section. One thing in common among these three data types is all these wrapper classes for data types are served as intermediate to access the underlying data members.

3.1 rstring

`char*` is a special data type which needs special treatments because a character string itself is a *SEXP* object in **R**, so one more layer of indirection has to be added to process `char*` data. The iterators of `rvector<rstring>` and `rmatrix<rstring>` are pointers to `rstring` object. Let `ptr` be an iterator of `rvector<rstring>` (a pointer to `rstring`).

1. The syntax to modify (or replace, to be precise) the character string being pointed is

   ```
   rvector<rstring>::iterator ptr = ...;
   *ptr = rstring("new␣string");
   ```

2. The syntax to extract the character string data from `ptr` is

   ```
   const char* str = ptr->c_str();
   ```

   In other words, `rstring` is used as an intermediate to access elements of a string vector without knowing the underlying **R** mechanism.

See `demo_str` for a simple example on how to manipulate string characters.

3.2 rdate

The class rdate is wrapper for **R** data type "Date". **R** is a flexible system that a "Date" object could be any numeric value (integer or real) as long as the value is valid: it causes ambiguities for programmers. Under the rcppbind framework, the wrapper class rdate is enforced to be integer values. A set of member functions are provided to access the date information contained in the given data; hence, the low-level implementation is hidden to library users. A list of member functions is as follows:

1. time(struct tm& tm): return the date info using a tm structure.
2. int operator-(const rdate&): return the date difference;
3. rdate operator+(int): forward a date object;
4. static void date2tm(int date, struct tm& tm), static int tm2date(const struct tm& tm): date and tm conversion.

3.3 rtime

The class rtime is wrapper for **R** datatime type "POSIXct". Time is always a confusing issue on most systems. In **R**, there are two classes for time variables: POSIXlt, POSIXct.

1. The POSIXct (a calendar-time in POSIX standard) is a numeric value internally, while POSIXlt (a local-time in POSIX standard) is indeed a list with 9 fields.

   The benefit of POSIXlt is that all time information can be directly accessed from those 9 fields. Its disadvantage is that we cannot have a vector of POSIXlt: there is no such thing called a vector of list in **R**. On the other hand, the compact form is just a numeric value (mostly real). We can put POSIXct in various data structures, but we need to compute time information from the numeric value.

   Indeed, the computational penalty brought by POSIXct is minimal.
2. Many times, we see date-time object in **R** with class attribute POSIXt. It is inherited from the above two classes to allow operations between POSIXct and POSIXlt.

Under the rcppbind framework, the wrapper class rtime is enforced to be real values – natural for POSIXct. A set of member functions are provided to access the time information contained in the object. A list of member functions is as follows:

1. void time(struct tm& tm)
2. double operator-(const rtime& obj): compute the time difference;
3. rtime operator+(double timediff): forward a time;
4. static void time2tm(double time, struct tm& tm): time to tm structure;
5. static double tm2time(const struct tm& tm): tm structure to time.

Notes

1. The timezone information of rtime is NOT implemented: the default timezone is always assumed to be “GMT”.

   There is a reason for doing so. Consider a **R** POSIXct object wrapped by rvector<rtime>. Timezone is actually an attribute of vector object, but not that of rtime objects. As an element of the datetime vector, a rtime is simply a real value internally.

4 Entrance & Exit

4.1 Entrance – Retrieving Information from R

In order to minimize the programmers' burden in retrieving **R** objects from the `.Call` interface, a set of rcppbind macros are defined to automatically wrap **R** objects into C++ objects under the `rcppbind` framework. The syntax of the interface is as follows:

```
RCPPn(module_name, T1, ..., Tn)
```

where `RCPPn` is a macro which defines a `rcppbind` module, *`module_name`*, with n parameters, and `T1, ..., Tn` are the types of input arguments. The *`module_name`* is the entry point not only to the `.Call` interface, but also to the function of real implementation. One typical example is as follows:

```
#include <rcppbind.h>

RCPP2(mymodule, double, rvector<double>);
SEXP mymodule(const double& t1, const rvector<double>& t2) { ...; }
```

Inside **R**, the `.Call` interface for calling this module is as follows:

```
mymodule <- function(x, y) .Call("mymodule", x, y)
```

Of course, the module has to be loaded before the **R** "`mymodule`" function could be called. It is also a good practice to ensure both `x, y` are of right types inside the **R** function `mymodule`.

Notes

- The current implementation only support 0 to 9 input arguments. If your module has more than 9 input parameters, please check the header file `rmacro.h` and define your own macro accordingly.
- The types `T1, ..., Tn` in the macro definition can only be wrapper classes: `rvector, rmatrix, rdataframe, rlist`, and some primitive types, such as `int, Rcomplex, char*, rdate, rtime`.
- All input arguments are declared as references to constants: the template library strictly enforces the **R** parameter-passing convention.
- The C++ exception mechanism is used in *`module_name`*. If an error occurs during the execution of the module, the program will exit nicely and return the control to **R**.

  Argument type checking is performed in the `RCPPn` macro to ensure that input arguments are of the right types. An exception will be thrown if the parameter type does not match.

4.2 Exit – Returning Objects to R

The real implementation function is of the form:

```
SEXP module_name(const ...);
```

The return type of the function is always `SEXP`. In `rcppbind`, all wrapper classes for class types, namely `rvector, rmatrix, rlist, rdataframe`, are derived from a common base-class `robject`, where an operator `SEXP` is defined to convert them implicitly to `SEXP`. Hence, under the template framework, developers only needs to return a `rcppbind` object in the end without concerning the *SEXP* issue.

Notes

- If one wants to return an integer value to **R**, one needs to return a vector of integer of length 1. The same to `double, char*`, etc.
- For more complicated data structures, one has to rely on `rlist`. It is recommended to manipulate class attribute inside the **R** function to the `.Call` interface.
- `R_NilValue` can be returned for nothing.

5 Summary

A quick summary of the `rcppbind` template library:

1. The only header file needs to be included in your project is `rcppbind.h`.
2. The module name in the `RCPP`*n* macro serves as entry point not only to the `.Call` interface, but also to the function of real implementation.
3. One needs to know both class and data types of an **R** object in order to bind it: some internal conversion might occur.
4. The C++ implementation function takes constant references as input arguments.
5. The template library only supports `.Call` interface.
6. The library supports four class types. More complicated classes can be derived from `list`.
7. Classes `rstring, rdate, rtime` are used as an intermediate to access `char*, Date, POSIXct` data respectively.
8. Return a `rcppbind` object to **R** at the end of the your code.
9. The index starts at 0 in C++, while 1 in **R**.

6 Acknowledgement

I would like to thank Dominick Samperi for developing and releasing the `rcpp` package. It has similar functionalities as mine, and I even considered the possibility of merging mine within his package. Eventually, since two designs are drastically different, I decide to release my code into a separate package instead. Nevertheless, many good ideas and codes are shamelessly stolen from Samperi directly.