Rough Draft: Not for Distribution

R/Cpp: Interface Classes to Simplify Using R Objects in C++ Extensions

James J. Java[*][†]
Time Warner Cable

Daniel P. Gaile
University at Buffalo

Kenneth F. Manly
University at Buffalo

July 23, 2007

Abstract

The R statistical environment provides a very utile programming language, some of whose utility derives from the implementation of R as an interpreted language. However, simulation or analysis can sometimes benefit from manipulating R data in an existing library or in optimized code written in a compiled programming language. Through the mechanism of foreign-language extensions (such as C++ or FORTRAN), computationally intensive code, for example, can be compiled in a lower-level language linked to the R environment, and have its results passed back to R. Such extensions are somewhat difficult to create, though.

This paper describes the development of a set of classes written in C++ to offer simplified ways of manipulating R objects in a C++ context. We hope to provide in **R/Cpp** the means for ready creation of fast extensions to R, without the typical concomitant difficulty in implementation.

1 The Programming Languages

1.1 R

R is an "integrated suite of software facilities for data manipulation, calculation and graphical display" designed around a true computer language[1]. It is an open-source GNU[2] project which is similar to the S language and environment developed at Bell Laboratories. (S-PLUS is a commercial version of S.) R provides many functions and packages implementing statistical techniques, and in fact many of its users consider R to be primarily a statistics system.

As a data-analysis *environment*, R provides[1,3]

- facility of data importation, manipulation, and storage;
- a large collection of tools for data analysis;
- highly customizable, production quality graphics;
- ready extensibility through user-defined functions and a growing list of add-on packages.

[*]Correspondence to: James J. Java, Time Warner Cable, 71 Mt. Hope Avenue, Rochester, New York 14620–1090, U.S.A.

[†]E-mail: `james.java@twcable.com`

As a programming *language*, R provides[3–5]

- control statements for conditional or repetitive execution;
- user-defined recursive functions;
- support for object-oriented constructs;
- operator overloading;
- interactive command interpretation and object manipulation;
- a simple interface to compiled code that has been linked to R.

The two lattermost language characteristics suggest a potential problem and its resolution, respectively. The flexibility provided by code interpretation allows for such conveniences as command-line interactivity, run-time object resolution, and fast debugging; but this flexibility can cost computational efficiency, and some perfectly valid programming structures cannot take advantage of R's built-in code optimizations, thus running slower than the same structures built in the idiom of a compiled language. R, though, allows its objects to be passed into compiled extensions linked to the R environment, where the objects can be invoked, used, altered, and then sent back where they came from, as needed. If some piece of R code plods along against the wishes of the programmer, its functionality can be farmed out to speedier C or FORTRAN libraries, for the best of both worlds.

1.2 R Language Extensions[5]

The R environment provides the functions `.C` and `.Fortran` as a standard interface to compiled code that has been linked into R, either at build time or through the `dyn.load` function. The `.C` function is intended for compiled C code, but it can be used with other languages that can create C interfaces, such as C++. (Likewise the intention for the interface functions `.Call` and `.External`, which are discussed below.)

The first argument to each function is a character string giving the symbol name known to C or FORTRAN, that is, the function or subroutine name. A further 65 arguments giving R objects can be passed to the compiled code. Normally these are copied before being passed in (unless the argument `DUP=F` is given), and copied again to an R list object when the compiled code returns. If the arguments are given names, these are used as names for the components in the returned list object.

The following table maps the modes of R vectors to the corresponding arguments passed to a C function or FORTRAN subroutine.

R storage mode	**C type**	**FORTRAN type**
`logical`	`int*`	`INTEGER`
`integer`	`int*`	`INTEGER`
`double`	`double*`	`DOUBLE PRECISION`
`complex`	`Rcomplex*`	`DOUBLE COMPLEX`
`character`	`char**`	`CHARACTER*255`
`raw`	`char*`	none

R also provides the functions `.Call` and `.External` as alternative interfaces to compiled C code that has been linked into R. Whereas `.C` allows type-checking (since an argument on one side of the interface must typewise match its partner on the other side) and duplication of its arguments,

`.Call` and `.External` transfer type-checking to the C code and do not automatically duplicate their arguments. Of the two `.Call` is simpler to use, but `.External` is more general.

A call to `.Call` is very similar to `.C`; for example,

```
.Call("hello", a, b)
```

The first argument should be a character string giving a C symbol name of code that has already been loaded into R. Up to 65 R objects can be passed as arguments. The C side of the interface is

```
#include <R.h>
#include <Rinternals.h>

SEXP hello(SEXP a, SEXP b)
...
```

A call to `.External` is identical,

```
.External("hello", a, b)
```

but the C side of the interface is different, having only one argument:

```
#include <R.h>
#include <Rinternals.h>

SEXP hello(SEXP args)
...
```

Here `args` is a `LISTSXP`, a Lisp-style list from which the arguments can be extracted.

In C code, all objects you deal with will be of type *SEXP*[†], which is a pointer to a structure with the type definition `SEXPREC`. This structure is a *variant type*, a sort of catch-all container which can handle the usual types of R objects—vectors of various modes, functions, environments, language objects, and so on. Access macros are used on `SEXP`s to get from them the basic data types that C can use directly. The R types and their corresponding `SEXP` type identifiers are provided in the following table:

SEXP type	**R equivalent**
`REALSXP`	numeric with storage mode `double`
`INTSXP`	integer
`CPLXSXP`	complex
`LGLSXP`	logical
`STRSXP`	character
`VECSXP`	list (generic vector)
`LISTSXP`	“dotted-pair” list
`DOTSXP`	a ‘`...`’ object
`NILSXP`	NULL
`SYMSXP`	name/symbol
`CLOSXP`	function or function closure
`ENVSXP`	environment

[†]*SEXP* is an acronym for *S*imple *EXP*ression, common in Lisp-like languages.

A complete but toy example, presented as a series of steps, will illustrate how `.External` is used. We assume that the operating system used to run the example has been configured for building add-on packages from source, as described by the R Development Core Team[6].

1. Create an empty directory to assemble and run the example project in. Place any new files discussed below into this directory.

2. Create the file `hello.c` and, using a text editor, add the following code to it:

```c
// hello.c

#include <R.h>
#include <Rinternals.h>
#include <Rdefines.h>

SEXP hello(SEXP args) {
  SEXP langNameSXP = CADR(args);// extract 1st SEXP argument passed to C
  char* languageName = CHAR(STRING_ELT(langNameSXP, 0));// extract string
  Rprintf("Hello from %s\n", languageName);// print message from R
  SET_STRING_ELT(langNameSXP, 0, COPY_TO_USER_STRING("C"));// make it "C"

  return R_NilValue;
}
```

3. Open a command window and navigate to the example directory. Compile `hello.c` and link it to R by issuing the command

   ```
   R CMD SHLIB hello.c
   ```

   If all goes well, the compiler and linker will output a few benign (i.e. without the word "error") diagnostic messages and the directory will now contain a file called either `hello.so` or `hello.dll`, depending upon your operating system.

4. Create the file `hello.R` and add the following code to it:

```r
# hello.R

# Load shared object/DLL compiled from C code:
dyn.load(paste("hello", .Platform$dynlib.ext, sep=""))
languageName <- "R"
SayHello <- function() {
  returnValue <- .External("hello", languageName) # call C function
  cat("Hello from", languageName, fill=T) # print message from C
}
SayHello()
dyn.unload(paste("hello", .Platform$dynlib.ext, sep="")) # unload
```

5. Start up an interactive R session and navigate to the example directory. Type

   ```
   source("hello.R")
   ```

to run the example project. It should produce the following output:

```
Hello from R
Hello from C
```

This is what happens, in an abstract way, when the example project is `source`d. R loads the shared object/DLL into memory, putting the C interface function `hello` into the symbol table so it can be invoked. We create the character variable `languageName` with the value '`R`', then use the `.External` function to pass `languageName` into the C function `hello`.

C receives the entire set of objects (only one object here, but this interface allows an unlimited number) from R as a single `LISTSXP`, `args`, which is itself a list of `SEXP` pointers to all the objects given as arguments to `.External`. The `SEXP` particular to `languageName` is extracted from `args` by the `CADR` macro and assigned to `langNameSXP`, from which in turn is extracted a `char*` string variable native to C; we also call the `char*` variable `languageName` to correspond to its native-R counterpart. Note that R's `languageName` actually contains C's `languageName`: R is written in C, and part of the `SEXPREC` structure of an R character vector is the simple `char*` strings supplying values to the vector's elements. And recall that arguments to `.External` are not automatically duplicated; through `langNameSXP`, then, we have direct control of the `languageName` variable within the R environment.

Next we print out a message containing the value of `languageName` assigned in R, then proceed to change that value by using macros contained in the R-supplied C header files `Rinternals.h` and `Rdefines.h`. The C function `hello` finishes up and returns a `NULL`. Back in R, we print out a message containing the value of `languageName` assigned in C and note the change in value. R unloads the shared object/DLL from memory and returns to interactive mode.

In this example, we have deliberately gone afoul of the recommendations of the R Development Core Team:

> Before you decide to use `.Call` or `.External`, you should look at other alternatives. First, consider working in interpreted R code; if this is fast enough, this is normally the best option. You should also see if using `.C` is enough.... The `.Call` and `.External` interfaces allow much more control, but they also impose much greater responsibilities so need to be used with care. Neither `.Call` nor `.External` copy [*sic*] their arguments. You should treat arguments you receive through these interfaces as read-only[5].

But the interfaces exist, and are quite powerful when used properly. In the **R/Cpp** package we try to take on most of the aforementioned "responsibilities" that make moving R objects into fast extensions perilous, by using certain features of C++ that allow the imported R objects to be manipulated much as if in their native environment.

1.3 C++

C++ can be considered a superset of the C programming language with extensions for object-oriented programming. Like C, C++ separates the core programming language from all input and output operations, and from the implementation of data types other than the inherent integer, character, and floating-point types. Input and output commands are implemented as program functions for other functions to call, which helps to minimize the number of platform-dependent features built into the C++ language and its compilers. In addition to the built-in data types, C++ includes a pointer data type to hold memory addresses from which variable values can be retrieved indirectly[7,8].

C++ uses the function as the basic unit of program execution. Every program begins execution in a function named `main` and terminates when the `main` function ends. The `main` function can call other functions, which can in turn call other functions. Programmers can build their own function libraries—collections of functions that can be called from within any program linking itself to the library[8].

The ANSI/ISO standard definition of C++[9] incorporates into itself the core C language and the Standard C Library. The Standard C Library defines a standard (that is, common to all C++ compilers) set of library functions that support streaming character input/output to the console and disk files; it also defines other standard libraries supporting common operations on characters, dates, times, strings, function arguments, and for math, memory allocation, and so on. Because C++ "contains" C, it augments the low-level memory management and computing efficiency of C with its own high-level programming efficiency.

Standard C++ also includes the Standard C++ Library, which consists of definitions of classes, a *class* being that object-oriented mechanism of C++ which allows the addition of user-defined data types to the language. Classes provide a way to encapsulate data and the functions that operate on that data, and more specific classes can be derived from those more generic (IS-A), or included in those of larger scope (HAS-A). Among the classes in the Standard C++ Library are classes that enable instantiation of:

- character string objects with string operations similar to those of other high-level languages;
- input/output stream objects that read and write character data to and from the console and disk files, and which are extensible to other input/output devices;
- container objects—such as vectors, lists, and queues—used to contain and manipulate collections of data objects;
- generic algorithms to perform operations on the container objects;
- numerical objects to represent complex numbers;
- localization objects to control at run time the display of dates, currency, and other culturally mutable signs, based on the geographical location of a program's user;
- exception objects to manage execution failure of a program gracefully[8].

Not directly instantiable themselves, but used to build generic data-type container objects, are C++ template classes. A prominent example of C++ template classes is the Standard Template Library, a set common containers and generic functions included in the Standard C++ Library.

1.4 The Standard Template Library

C++ templates are used to define generic container classes for managing sets of data objects, and generic functions that take parameterized arguments. A part of the Standard C++ Library, the Standard Template Library (STL) is a library of container class templates, generic iterators, and algorithmic function templates. STL containers are agnostic of the objects they contain; neither should the objects concern themselves with the details of containment. The containers are simply objects that hold other objects, and because container classes are templatized, they can hold any type of object, including those instantiated from user-defined classes. *Iterators* are pointer-like objects that allow movement from one element of a container to the next, a necessity for containers whose elements sit in non-contiguous memory. *Algorithms* operate on containers by modifying, copying, removing, or otherwise manipulating their elements[10,11].

A simple example will illustrate how the STL is used in C++. We assume that the operating system used to run the example has been configured for building add-on packages to R from source, as described by the R Development Core Team[6].

1. Create the file `stl.cpp` and, using a text editor, add the following code[†] to it:

```
// stl.cpp

#include <iostream>
#include <vector>
#include <string>
#include <iterator>
using namespace std;

int main() {
  // Create empty vector for strings:
  vector<string> hello;

  // Reserve memory for six elements to avoid reallocation:
  hello.reserve(6);

  // Append some elements:
  hello.push_back("Hello,");
  hello.push_back("world");
  hello.push_back("--");
  hello.push_back("from");
  hello.push_back("the");
  hello.push_back("STL");

  cout << "size(): " << hello.size() << endl;

  // Print elements using an iterator loop:
  vector<string>::iterator i;
  for (i=hello.begin(); i!=hello.end(); ++i)
    cout << *i << " ";
  cout << "!" << endl;

  // Print elements using the 'copy' algorithm:
  copy(hello.begin(), hello.end(), ostream_iterator<string>(cout," "));
  cout << "!" << endl;

  // Remove the last four elements:
  hello.erase(hello.begin()+2, hello.end());
  cout << "size(): " << hello.size() << endl;
```

[†]We recommend in general not using the `using namespace` statement, as it contravenes the reason for defining `namespace`s in the first place. We use it here, though, to simplify the code for presentation.

```
39    copy(hello.begin(), hello.end(), ostream_iterator<string>(cout," "));
40    cout << "!" << endl;
41
42    return 0;
43  }
```

2. Open a command window and navigate to the example directory. Compile `stl.cpp` by issuing the command

   ```
   g++ -o stl stl.cpp
   ```

   If all goes well, the compiler and linker will output some non-error diagnostic messages and the directory will now contain an executable file called either `stl` or `stl.exe`, depending upon your operating system.

3. From the same command window, type the name of the executable file and press `Enter`. The following output should appear:

   ```
   size(): 6
   Hello, world -- from the STL !
   Hello, world -- from the STL !
   size(): 2
   Hello, world !
   ```

A few glosses on this code:

- **L. 11** instantiates the vector container object `hello` for holding character string objects. A STL `vector` is an all-purpose container in that it can be readily grown and shrunk, its elements can be iterated through bi-directionally or selected directly, and it acts as a dynamic replacement for the sometimes difficult C++ arrays inherited from C.

- **L. 14** reserves enough memory for `hello` to hold six elements, so that as objects are added to the originally empty vector, memory reallocation and copying because of insufficient contiguous memory become unnecessary.

- **Ll. 17–23** append a full sentence of single string objects on to the originally empty `hello` vector.

- **L. 27** creates a `vector<string>` iterator which, in **ll. 28–29**, gets an initial value pointing to the first element of the `hello` vector, and which is then incremented by the `for` loop until each element of `hello` has been printed out.

- **L. 33** also prints out each element of `hello`, but this time via the `copy` algorithm.

- **L. 37** removes the last four elements of `hello`. Note that adding n to the random-access iterator returned by `hello.begin()` points the resulting iterator to the $(n + 1)$th element of the `hello` vector.

In **R/Cpp**, we have decided that the R-object containers formed in our C++ extensions should act as much like STL-compliant containers as possible while still retaining R-like functionality. Then users of **R/Cpp** have the best of both worlds: easily manipulable R-like objects that can also be treated like STL containers, for use within a computationally efficient C++ environment.

2 R/Cpp

2.1 Development

R/Cpp developed from a simulation problem in computational genetics. When we had realized that we must move a certain amount of the simulation code from R to a fast extension linked to R, the next problem became what language we ought to write the extension in. FORTRAN was considered but dismissed as unable to receive or manipulate R `list` objects; that pretty much left us to use C through the `.Call` or `.External` functions in R. Since `.External` doesn't require argument matching on the C side (recall from Section 1.2 that the arguments are sent from R as a single "dotted-pair" list), and because C++ can generate C interfaces and is a "better" programming language than C, we decided on C++ through `.External` as potentially leading to the most generic and reusable code.

After some ill-fated (and slow!) tests using GNU-licensed variant classes to work with R objects in C++, we decided that STL-like containers might represent a better approach to the problem, and so we created new template classes. Because of their apparent usefulness, we broke out the template classes as **R/Cpp** for general use elsewhere. Debugging continued as some maddeningly subtle errors plagued **R/Cpp**, but these were overcome (as far as we know), and we have used the **R/Cpp** classes quite successfully since then.

Further development continues with a new wish to make the **R/Cpp** containers fully STL compliant, if that is even possible. (See the discussion in Section 2.9 for details.)

2.2 Implementation

The core of **R/Cpp** is about 2000 lines of modern C++ code that defines four major classes: `RcallArgs`, `vector_r<>`, `array_r<>`, and `RObject`. Development was done primarily on Windows XP using Visual Studio .NET 2003 (and more recently Visual Studio 2005), but the code was also regularly compiled with gcc and run on Windows XP and on various *nixes (including Red Hat Linux and Solaris) to ensure platform independence comparable to R's platform independence. The theological-type debates regarding the standards-compliance of the various C++ compilers never really influenced the development of **R/Cpp**, with universal warning- and error suppression being the prime movers for multiple-platform testing. (We did need to tailor the code extensively a couple of times for it to compile and run on all platforms, though care was taken not to rely on any ephemeral characteristics of the compilers just to avoid diagnostic complaints.)

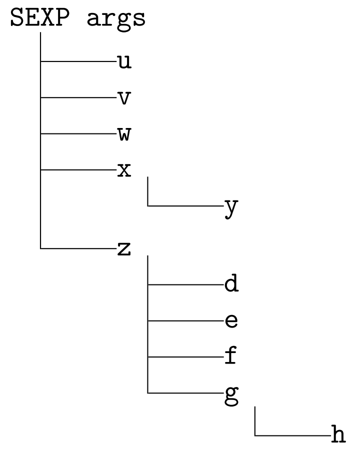
The four major classes of **R/Cpp** allow simplified manipulation of R objects within C++, and we describe them now.

2.3 The `RcallArgs` class

`RcallArgs` is a class for dealing easily with the argument pairlist sent over to C++ by `.External`. It is a current requirement of **R/Cpp** (and of the R-extensions mechanism in general) that users know the structure of R objects they pass to C++; `RcallArgs` takes advantage of that knowledge to easily extract individual objects from the argument pairlist. The pairlist shows up in C++ as a single `SEXP` (a pointer to a `SEXPREC` object), whose pointee contains other `SEXP`s pointing to the individual objects sent through the R interface; if any of those objects are vector `list`s, then they too contain `SEXP`s pointing to the objects that they themselves contain, and so on, *ad infinitum*, theoretically. Suppose that the following objects are created in an R environment and passed to a linked and loaded C++ extension containing a function called `test`:

```
u <- 1:5 # integer vector
v <- c(u, 3.14) # numeric vector
w <- c("This", "is", "an", "’RcallArgs’", "test") # character vector
x <- list(y=as.logical(0:4)) # F T T T T
z <- list(d=u, e=v, f=w, g=list(h=as.logical(0:4)))
...
returnValue <- .External("test", u, v, w, x, z)
```

In the C++ function `test`, the arguments passed from R now have the following hierarchy:



That is, `y` is “in” `x`; `d`, `e`, `f`, and `g` are in `z`, and everything is in `args`. The `RcallArgs()` constructor creates an object of class `RcallArgs` from the argument pairlist,

```
SEXP test(SEXP args) {
RcallArgs rca(args);
...
```

and the overloaded parenthesis operator `()` allows access to the pairlist’s contained objects based on the internal hierarchy described above, using zero-based natural numbering to identify an object’s position in each level of the hierarchy. In the example, the first tier of objects under `args` comprises `u`, `v`, `w`, `x`, and `z`, which are indexed in `RcallArgs` by 0–4, repectively. Separate second tiers of objects exist under each of `x` and `z`; `x` contains `y`, which is indexed by 0, and `z` contains `d`, `e`, `f`, and `g`, which are indexed by 0–3. A solitary third tier exists under `g`, which contains `h`, indexed by 0.

To get the `SEXP` pointing to a particular object, the `RcallArgs` parenthesis operator takes the indices that trace through the hierarchy a path that leads to the object. Retrieving an object pointer from the first tier requires one argument to `operator ()`; retrieving an object pointer from the second tier requires two arguments; third tier, three arguments; and so forth. To get, say, the `SEXP` for `u` from the `RcallArgs` object `rca`, the call is

```
SEXP sexp_u = rca(0, -1);
```

Well, there is actually more than one argument needed to retrieve a first-tier pointer, but the second argument has nothing to do with tracing the object hierarchy. In C++ (through inheritance from C), a function that takes a variable number of arguments, like `operator ()` in `RcallArgs`, needs to know when its argument list is complete; here we use `-1` as an ending delimiter to tell `operator ()` that the previous argument is the last one relevant to this function call. The function call to get to `y` is, then,

```
SEXP sexp_y = rca(3, 0, -1);
```

and that to get to `h` is

```
SEXP sexp_h = rca(4, 3, 0, -1);
```

The `SEXP`s extracted from argument pairlists can be used in R-provided macros to manipulate their objects, but we have taken further steps to make using R objects in C++ more familiar to users of R to whom C++ is less familiar: the object `SEXP`s extracted by `RcallArgs` can be passed into constructors of the **R/Cpp** classes `vector_r<>` or `array_r<>` to make the corresponding objects much easier to work with.

2.4 The `vector_r<>` Class

`vector_r<>` is a template class for wrapping around the `SEXP`s of R objects passed into C++ extensions, and for directly building new R objects in C++ extensions. The most commonly used objects within the R environment are vectors of various storage modes. In this context a *vector* is analogous to its mathematical counterpart, the one-dimensional array, with elements that can hold a single type from among several numerical types and a character-string type. These types, the so-called atomic storage modes in R, are `logical`, `integer`, `numeric`, `complex`, `character`, and `raw`. Each mode can be invoked in a vector in R by using the `vector` function,

```
vector(mode = storage_mode , length = length)
```

where *`storage_mode`* is one of the six atomic storage modes; each mode also has a function with the same name as itself for creating new vectors, as in, for example,

```
logical(length = length)
```

The other types have corresponding functions that also take a `length` argument.

Using the `vector_r<>` class to contain R vector objects passed into C++ aids in their use and manipulation in two primary ways. First, it makes getting and setting individual elements quite simple (rather simpler than getting or setting with the macros provided by R); second, as a pseudo-STL (Standard Template Library; v. Section 1.4) container, `vector_r<>` allows the programmer to take advantage of the various STL algorithms available for expediting certain commonly desired operations on its elements.

For example, if the following R object is sent through `.External` to a C++ extension,

```
u <- 1:5 # integer vector
```

it can be received in C++ as a `SEXP` and governed by `vector_r<>` like this:

```
vector_r<int> u(rca(0, -1));
```

where the `RcallArgs` object `rca` has supplied `vector_r<>` with the `SEXP` pointing to R's `u`. Now `u` in C++ is a `vector_r<>` object, and gets and sets on it can be performed as follows:

```
// In R, 'print'ing u gives [1] 1 2 3 4 5
int singleElement = u[1]; // singleElement == 2 is true
u[0] = 666; // In R, 'print'ing u would give [1] 666 2 3 4 5
```

Note that `vector_r<>` uses zero- rather than one-based indexing like R, which is customary in C++. Note also the template parameter `int` passed to `vector_r<>`: it is required that `vector_r<>` receive as a parameter a C++ data type according with the storage mode of the R object that it's meant to contain. The R storage modes are matched to their appropriate **R/Cpp**–C++ data types in the following table:

R storage mode	C++ data type
`logical`	`bool`
`integer`	`int`
`numeric`	`double`
`complex`	`Rcomplex`[†]
`character`	`char*`
`raw`	`unsigned short int`[†]

Using the bracket operator `[]` on a `vector_r<>` object extracts from it the single element at the given index, returned as the appropriate C++ data type from the table above; as in,

```
int singleElement = u[1];
```

Using the bracket operator along with the assignment operator `=` in the following way sets the single element at the given index to a new value; as in,

```
u[0] = 666;
```

The **R/Cpp** containers `vector_r<char*>` and `vector_r<SEXP>`, corresponding to a vector of R storage mode `character` and to an R generic vector, respectively, act somewhat differently. While the bracket operator still gets elements from `vector_r<>`s of these types, it no longer works as part of an assignment operation to set elements to new values. The reason for the breakage has to do with the fact that elements of R `character` vectors and of R generic vectors are not stored in contiguous memory, whereas the elements of the other R vectors are. The current solution is to provide a `set` member function in the class specializations of the `vector_r<char*>` and the `vector_r<SEXP>` containers, to be used like this:

```
# In R:
w <- c("This", "is", "a", "'vector_r<>'", "test")
x <- list(y=as.logical(0:4))
...
// In C++:
w.set("hack", 4); // changes "test" to "hack"
x.set(w.GetObject(), 0); // makes 'w' the first element of 'x'
```

Though this solution works, it is aesthetically displeasing (it breaks the *orthogonality* of the `vector_r<>` class) and doesn't jibe with how container elements arranged in non-contiguous memory are handled in the STL. A better solution would probably be to move `char*`- and `SEXP`-vector handling to a separate **R/Cpp** `list_r<>` class which deals with the elements of its objects in the same manner in which STL `lists` deal with theirs. In fact, we have decided to make this change, along with beefing up the iterators of the `vector_r<>` class, in order to put **R/Cpp** containers as

[†]Unfortunately, we have not yet implemented **R/Cpp** analogues to the `complex` and `raw` storage mode vectors ☹. The situation will change in the next revision of the software; see Section 2.9 for information on this and other imminent updates.

close as possible to their true STL counterparts. The change will happen in a future revision of **R/Cpp**.

A further comment on `vector_r<SEXP>`. The `vector_r<SEXP>` container is used to wrap R generic vectors, or `lists`, whose elements can hold any type of R object, with a single `list` possibly containing many different types of R objects! In C++ every R object is represented by a `SEXP` pointer which points to a `SEXPREC` structure, which in turn "holds" the data of the R object. An R list is really just a container full of `SEXP`s pointing to the contained objects; therefore in **R/Cpp** we must treat its elements as `SEXP`s, hence the `SEXP` template parameter given to `vector_r<SEXP>`. The `vector_r<>` class provides the member function `GetObject` to return the `SEXP` of an enwrapped R object; a `vector_r<SEXP>` object can take that `SEXP` and use it to contain within itself the corresponding R object; as in, from the code pieces just above,

```
x.set(w.GetObject(), 0); // makes 'w' the first element of 'x'
```

where element 0 of the `vector_r<SEXP> x` is set to the `character` vector `w`'s `SEXP` (as returned by `GetObject`), in effect embedding `w` in `x`.

* * *

Recall from Section 1.4 that the Standard Template Library algorithms operate on containers by modifying, copying, removing, or otherwise manipulating their elements[11]. Algorithms are not member functions of the container classes. Instead, they are global functions that operate with iterators. The advantage of this is that instead of each algorithm being implemented for each container type, all are implemented only once for any container type. Algorithms will also operate on user-defined container types[10]. This concept allows the programmer to take advantage of the power and flexibility of the STL algorithms while using **R/Cpp** `vector_r<>` containers.

Here is a partial list[10] of STL algorithms tested with `vector_r<>` containers:

Non-modifying Algorithms

Algorithm Name	Effect
`for_each()`	Performs an operation for each element
`count()`	Returns the number of elements
`count()_if()`	Returns the number of elements that match a criterion
`min_element()`	Returns the element with the smallest value
`max_element()`	Returns the element with the largest value
`find()`	Searches for the first element with the passed value
`find_if()`	Searches for the first element that matches a criterion
`search_n()`	Searches for the first n consecutive elements with certain properties
`search()`	Searches for the first occurrence of a subrange

Modifying Algorithms

Algorithm Name	Effect
`for_each()`	Performs an operation for each element
`copy()`	Copies a range starting with the first element
`fill()`	Replaces each element with a given value
`fill_n()`	Replaces n elements with a given value
`generate()`	Replaces each element with the result of an operation
`generate_n()`	Replaces n elements with the result of an operation
`replace()`	Replaces elements that have a special value with another value
`replace()_if()`	Replaces elements that match a criterion with another value

Removing Algorithms

Algorithm Name	Effect
`remove()`	Removes elements with a given value
`remove_if()`	Removes elements that match a given criterion
`remove_copy()`	Copies elements that do not match a given value
`remove_copy()_if()`	Copies elements that do not match a given criterion
`unique()`	Removes adjacent duplicates (elements that are equal to their predecessor)
`unique_copy()`	Copies elements while removing adjacent duplicates

Mutating Algorithms

Algorithm Name	Effect
`reverse()`	Reverses the order of the elements
`rotate()`	Rotates the order of the elements
`next_permutation()`	Permutates the order of the elements
`prev_permutation()`	Permutates the order of the elements
`random_shuffle()`	Brings the elements into a random order

Sorting Algorithms

Algorithm Name	Effect
`sort()`	Sorts all elements
`stable_sort()`	Sorts while preserving order of equal elements
`partial_sort()`	Sorts until the first n elements are correct
`partial_sort_copy()`	Copies elements in sorted order

Numeric Algorithms

Algorithm Name	Effect
`accumulate()`	Combines all element values (processes sum, product, and so forth)
`inner_product()`	Combines all elements of two ranges
`adjacent_difference()`	Combines each element with its predecessor
`partial_sum()`	Combines each element with all of its predecessors

(*N.B.* We have implemented some of these algorithms as member functions of `vector_r<>` out of necessity, but with the upgrade to heartier iterators in `vector_r<>` we will be able to remove most of them and allow the corresponding STL algorithms to take over. The modifying algorithms currently *do not* work with `vector_r<char*>` and `vector_r<SEXP>`, for reasons given in Section 2.4.)

The availability of these algorithms for `vector_r<>` simplifies operations on R objects in C++. Take as an example a function in C++ for finding the cumulative sums along the elements of an R vector; the code should receive a single numeric vector and output the result as another vector containing the cumulative sums. Using the R-supplied macros will produce code like the following:

```
RCPP_API SEXP cumsum_rext(SEXP args) {
  SEXP a = CADR(args);

  SEXP cumsum;
  PROTECT(a = AS_NUMERIC(a));
  int na = LENGTH(a);
  PROTECT(cumsum = NEW_NUMERIC(na));
  double* xa = NUMERIC_POINTER(a);
  double* xcs = NUMERIC_POINTER(cumsum);
  xcs[0] = xa[0];
  for (int i = 1; i < na; i++)
    xcs[i] = xcs[i-1] + xa[i];

  UNPROTECT(2);
  return(cumsum);
}
```

Using **R/Cpp** will make the code a little more concise:

```
RCPP_API SEXP cumsum_rcpp(SEXP args) {
  RcallArgs rca(args);

  vector_r<double> a(AS_NUMERIC(rca(0, -1)));
  vector_r<double> cumsum(a.size());
  partial_sum(a.begin(), a.end(), cumsum.begin());

  return cumsum.GetObject();
}
```

*   *   *

Finally, we wish to mention something about `vector_r<>` that may already be apparent from some of the code samples in this section. The `vector_r<>` class is used not only for wrapping R objects passed into a C++ extension, but also for creating completely new R objects that can be passed back to the calling R environment. In the **R/Cpp** code from the latest example, which finds the cumulative sums along the elements of an R vector, we create a new `vector_r<>` named `cumsum` to receive the cumulative sums, which then gets sent back to R:

```
  vector_r<double> cumsum(a.size());
  ...
  return cumsum.GetObject();
```

The `vector_r<>` constructor for creating new R vectors takes an argument of type `int`, which gives the length of the new vector.

2.5 The array_r<> Class

In computing, *row-major order* and *column-major order* describe ways to store multi-dimensional arrays in linear memory. Understanding array ordering is critical to passing arrays between programs written in different languages.

Row-major storage has a multi-dimensional array arranged in linear memory such that rows are stored one after the other. This is the approach taken by C, C++, and many other languages. In an array stored in row-major order, the differences between addresses of array elements in increasing rows is larger than the addresses of elements in increasing columns. For example, the following 2×3 array,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix},$$

can be declared in C++ by

```
int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

This lays out the array in linear memory thus:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}.$$

The linear offset from the beginning of the array to any element `A[`*`row`*`][`*`col`*`]` can be computed from

```
offset = row * num_cols + col;
```

where *num_cols* is the number of columns in the array.

In column-major storage, an array is stored in linear memory such that columns are stored one after the other. This is the approach taken by FORTRAN and—more important for this discussion—R. In an array stored in column-major order, the difference between addresses of array elements in increasing columns is larger than the addresses of elements in increasing rows. The example array already given,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix},$$

can be declared in R by

```
A <- array(c(1,4,2,5,3,6), dim=c(2,3))
```

This lays out the array in linear memory as

$$\begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 6 \end{bmatrix}.$$

with columns listed first. The linear offset from the beginning of the array to any element A[*row*,*col*] can be computed from

```
offset <- row + col * num_rows
```

where *num_rows* is the number of rows in the array.

These concepts can be generalized to arrays of more than two dimensions[12].

In **R/Cpp**, we manage both the passing of arrays from column-major R to row-major C++, and the generalization of R's multi-dimensional arrays into a C++ class, since C++ has no particularly good built-in mechanism for working easily with any sort of array.

In R, an array is just a vector with an *attribute* called dim, which itself is a vector containing the ordered dimensions of the array. The code to create a 2×3 array is, for example,

```
A <- array(c(1,4,2,5,3,6), dim=c(2,3))
```

whose representation is typeset as

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

The elements of A can be retrieved and set through the bracket operator by using one-based indexing:

```
a23 <- A[2, 3] # 'a23' equals 6
A[2, 3] <- 666
a23 <- A[2, 3] # 'a23' equals 666
```

In **R/Cpp**, array_r<> is a derived class of vector_r<> which adds to its superclass a dimensional attribute, and member functions for navigating through R's column-major arrays. All the vector_r<> operations can be performed on an object of class array_r<>, and the elements of the object set or retrieved through the parenthesis operator:

```
RcallArgs rca(args);
array_r<int> A(rca(0, -1));
int a23 = A(2, 3); // 'a23' equals 6
A(2, 3) = 666;
a23 = A(2, 3); // 'a23' equals 666
```

Included in the class is the member function subset for extracting subsets of array_r<>s.

2.6 The RObject Class

The `RObject` class allows the creation of R objects from a standard set of C++ types. For example, the constructor to create from a single C++ `bool` object an R `logical` vector with one element is given within this sample code:

```
# define FALSE 0
bool isCpp = true;
RObject logicalObject(isCpp);
vector_r<bool> isNotR(logicalObject()); // operator () returns RObject's SEXP
isNotR[0] = FALSE;
```

Note that using the parenthesis operator of an `RObject` with no argument returns the `SEXP` of its enclosed R object.

The following table lists the C++ types that `RObject` takes and their corresponding R-object conversions.

C++ type	**R vector**
`bool`	`logical` (single element)
`std::vector<bool>`	`logical`
`int`	`integer` (single element)
`std::vector<int>`	`integer`
`double`	`numeric` (single element)
`std::vector<double>`	`numeric`
`char*`	`character` (single element)
`std::vector<char*>`	`character`
`std::string`	`character` (single element)
`std::vector<std::string>`	`character`

2.7 Installation

Though **R/Cpp** could be installed as a (rather pointless) stand-alone package in R, its real value comes from its incorporation into packages that need the computational efficiency of C++. We have made the **R/Cpp** source files available through the following Web links:

- `http://www.priscian.com/code/Cpp_0.9.6.tar.gz`
- `http://www.priscian.com/code/Cpp_0.9.6.zip`

In what follows we assume that C++-extension code consists of a *header file* (whose name ends in `.h` or `.hpp`) and a *source file* (whose name ends in `.cpp`). Assuming that the operating system used to run the example has been configured for building add-on packages to R from source, as described by the R Development Core Team[6], we enumerate below the steps required to add **R/Cpp** functionality to an R project called `test`.

1. Create an empty directory to assemble and run the `test` project in. Place the **R/Cpp** files listed below into this directory:

   ```
   RCpp.h
   RCpp.cpp
   vcpp.h
   ```

```
common.h
algostuff.h
Rinterface.h
Rinterface.cpp
miscellaneous.h
miscellaneous.cpp
```

Also, place any new files discussed below into this directory.

2. Minimally, three user-made files are necessary: `test.R`, `test.h`, and `test.cpp`; they should be created with a favorite text editor.

3. The file `test.h` must include `RCpp.h` and declare any extension function using the "C" linkage specification:

   ```
   #include "RCpp.h"

   extern "C" {

   RCPP_API SEXP test_rext(SEXP args);

   }
   ...
   ```

   Function declarations should have the same *signature* (parameters, return type, storage-class attribute) as function definitions (see the following step).

4. The file `test.cpp` must include `test.h`; and any extension function must take a single `SEXP` argument, and it must be prefixed by the `RCPP_API` storage-class attribute:

   ```
   #include "test.h"

   RCPP_API SEXP test_rext(SEXP args) {
   ...
   ```

5. Open a command window and navigate to the example directory. Compile `test.cpp` and link it to R by issuing the command

   ```
   R CMD SHLIB test.cpp
   ```

   If all goes well, the compiler and linker will issue some error-free diagnostic messages and the directory will now contain a file called either `test.so` or `test.dll`, depending upon the operating system in use.

6. The file `test.R` should contain functions to load and unload the `test` shared object/DLL, and `.External` calls to any extension functions declared in `test.h`:

   ```
   # Load shared object/DLL compiled from C++ code:
   dyn.load(paste("test", .Platform$dynlib.ext, sep=""))
   Test <- function(x)
   ```

```
  .External("test_rext", x) # call C++ function
...
dyn.unload(paste("test", .Platform$dynlib.ext, sep="")) # unload
```

Add any other desired functionality to `test.R`.

7. Start up an interactive R session and navigate to the project directory. Type

   ```
   source("test.R")
   ```

   to run the example project.

2.8 Example Usage

We now present an entire small **R/Cpp** project based on the cumulative-sum example in Section 2.4 and the installation steps from Section 2.7. We have made the `cumsum` and base **R/Cpp** source files available through the following Web links:

- `http://www.priscian.com/code/cumsum.tar.gz`
- `http://www.priscian.com/code/cumsum.zip`

1. Create an empty directory named `cumsum` to assemble and run the `cumsum` project in. Place the **R/Cpp** files listed below into this directory:

   ```
   RCpp.h
   RCpp.cpp
   vcpp.h
   common.h
   algostuff.h
   Rinterface.h
   Rinterface.cpp
   miscellaneous.h
   miscellaneous.cpp
   ```

   Also, place any files discussed below into this directory.

2. Create the file `cumsum.h` and add the following code to it:

```
// cumsum.h

#ifndef __CUMSUM_H__
#define __CUMSUM_H__

#include "RCpp.h"

extern "C" {

RCPP_API SEXP cumsum_rext(SEXP args);

```

```
RCPP_API SEXP cumsum_rcpp(SEXP args);

}

#endif // !__CUMSUM_H__
```

3. Create the file `cumsum.cpp` and add the following code to it:

```
// cumsum.cpp

#include "cumsum.h"
using namespace std;

RCPP_API SEXP cumsum_rext(SEXP args) {
  SEXP a = CADR(args);

  SEXP cumsum;
  PROTECT(a = AS_NUMERIC(a));
  int na = LENGTH(a);
  PROTECT(cumsum = NEW_NUMERIC(na));
  double* xa = NUMERIC_POINTER(a);
  double* xcs = NUMERIC_POINTER(cumsum);
  xcs[0] = xa[0];
  for (int i = 1; i < na; i++)
    xcs[i] = xcs[i-1] + xa[i];

  UNPROTECT(2);
  return(cumsum);
}

RCPP_API SEXP cumsum_rcpp(SEXP args) {
  RcallArgs rca(args);

  vector_r<double> a(AS_NUMERIC(rca(0, -1)));
  vector_r<double> cumsum(a.size());
  partial_sum(a.begin(), a.end(), cumsum.begin());

  return cumsum.GetObject();
}
```

4. Open a command window and navigate to the `cumsum` directory. Compile `cumsum.cpp` and link it to R by issuing the command

   ```
   R CMD SHLIB cumsum.cpp
   ```

   If there are any error messages from the compiler or linker, check to see whether you have correctly followed all the steps thus far.

5. Create the file `cumsum.R` and add the following code to it:

```
# cumsum.R

# Load shared object/DLL compiled from C++ code:
dyn.load(paste("cumsum", .Platform$dynlib.ext, sep=""))

Cumsum1 <- function(x)
  .External("cumsum_rext", x) # call C++ function

Cumsum2 <- function(x, y)
  .External("cumsum_rcpp", x) # call C++ function

a <- 1:100
stm1 <- system.time(for (i in 1:100000) v1 <- Cumsum1(a))
stm2 <- system.time(for (i in 1:100000) v2 <- Cumsum2(a))
stm3 <- system.time(for (i in 1:100000) v3 <- cumsum(as.numeric(a)))

print(stm1)
print(stm2)
print(stm3)

dyn.unload(paste("cumsum", .Platform$dynlib.ext, sep="")) # unload
```

6. Start up an interactive R session and navigate to the `cumsum` project directory. Type

   ```
   source("cumsum.R")
   ```

   to run the project.

The R code calls each of the two cumulative-sum functions in the C++ extension, and then R's built-in `cumsum` function, 100000 times apiece and reports back the CPU time used *in toto* by the repeated calls to each function. The output will look something like this:

```
[1] 0.69 0.00 0.69   NA   NA
[1] 1.01 0.00 1.03   NA   NA
[1] 2.16 0.00 2.20   NA   NA
```

Here the **R/Cpp** code comes in second after the macro'd R-extension code, because there is some overhead associated with class-creation in R/Cpp that cannot be avoided. We insist that it has still has the other benefits detailed in the rest of this document!

2.9 Future Expansions

- Move `vector_r<char*>` and `vector_r<SEXP>` to a new `list_r<>` class that behaves more like STL lists.
- Implement copy-on-write semantics in `vector_r<>` and `list_r<>`.
- Move all R macros contained in **R/Cpp** to the smallest number of places possible (in case the R header files change).

- Hunt down various possibilities for optimization in the classes' code. We suspect that there may exist a few big opportunities for further speeding up **R/Cpp**.
- Complete the **R/Cpp** analogues to R's `complex` and `raw` storage mode vectors.
- Make the **R/Cpp** classes as fully STL-compilant as possible. They may never be *truly* STL-compliant because they are not agnostic of the objects they contain (which must be R objects), but they will be very close; and, what's more, they will be compatible with most of the STL algorithms.
- Introduce exception handling into the **R/Cpp** classes. This is currently lacking in **R/Cpp**.
- Create full class documentation of **R/Cpp**.

References

[1] The R Foundation. The R project for statistical computing. Web site, 2007. `http://www.r-project.org/`.

[2] The Free Software Foundation. The GNU project. Web site, 2007. `http://www.gnu.org/`.

[3] R Development Core Team. *An Introduction to R*. R Foundation for Statistical Computing, Vienna, Austria, 2007. URL `http://cran.r-project.org/doc/manuals/R-intro.pdf`.

[4] R Development Core Team. *R Language Definition*. R Foundation for Statistical Computing, Vienna, Austria, 2007. URL `http://cran.r-project.org/doc/manuals/R-lang.pdf`.

[5] R Development Core Team. *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2007. URL `http://cran.r-project.org/doc/manuals/R-exts.pdf`.

[6] R Development Core Team. *R Installation and Administration*. R Foundation for Statistical Computing, Vienna, Austria, 2007. URL `http://cran.r-project.org/doc/manuals/R-admin.pdf`.

[7] S. Satir and D. Brown. *C++: the Core Language*. O'Reilly & Associates, Inc., first edition, 1995. ISBN 156592116X.

[8] A. Stevens. *Teach Yourself C++*. IDG Books Worldwide, Inc., sixth edition, 2000. ISBN 0764546341.

[9] JTC1/SC22/WG21 Working Group. *Information Technology — Programming Languages — C++*. ANSI/ISO, second edition, October 2003. Reference number ISO/IEC 14882:2003(E).

[10] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison–Wesley Professional, 1999. ISBN 0201379260.

[11] H. Schildt. *The Art of C++*. McGraw–Hill/Osborne, 2004. ISBN 0072255129.

[12] Wikipedia. Row-major order — wikipedia, the free encyclopedia. Online, accessed April 10, 2006. `http://en.wikipedia.org/wiki/Row-major_order`.