Rcpp: Seamless R and C++ integration

by Dirk Eddelbuettel and Romain François

Abstract The **Rcpp** package simplifies integrating C++ code with R. It provides a consistent C++ class hierarchy that maps various types of R objects (vectors, functions, environments, ...) to dedicated C++ classes. Object interchange between R and C++ is managed by simple, flexible and extensible concepts which include broad support for C++ STL idioms. C++ code can be compiled, linked and loaded on the fly. Flexible error and exception code handling is provided. **Rcpp** substantially lowers the barrier for programmers wanting to combine C++ code with R.

Introduction

R is an extensible system. The 'Writing R Extensions' manual (R Development Core Team, 2010a) describes in detail how to augment R with compiled code, focusing mostly on the C language. The R API described in 'Writing R Extensions' is based on a set of functions and macros operating on `SEXP`, the internal representation of R objects. In this article, we discuss the functionality of the **Rcpp** package, which simplifies the usage of C++ code in R. Combining R and C++ is not a new idea, so we start with a short review of other approaches and give some historical background on the development of **Rcpp**.

The **Rcpp** package combines two distinct APIs. The first—which we call 'classic **Rcpp** API'—exists since the first version of **Rcpp**. While still contained in the package to ensure compatibility, its use is otherwise deprecated. All new development should use the richer second API. It is enclosed in the **Rcpp** C++ namespace, and corresponds to the redesigned code base. This article highlights some of the key design and implementation choices of the new API: lightweight encapsulation of R objects in C++ classes, automatic garbage collection strategy, code inlining, data interchange between R and C++ and error handling.

Several examples are included to illustrate the benefits of using **Rcpp** as opposed to the traditional R API. Many more examples are available within the package, both as explicit examples and as part of the numerous unit tests.

Historical Context

Rcpp first appeared in 2005 as a contribution (by Samperi) to the **RQuantLib** package (Eddelbuettel and Nguyen, 2010) before becoming a CRAN package in early 2006. Several releases (all by Samperi) followed in quick succession under the name **Rcpp**. The package was then renamed to **RcppTemplate**; several more releases followed during 2006 under the new name. However, no new releases were made during 2007, 2008 or most of 2009. Following a few updates in late 2009, it has since been withdrawn from CRAN.

Given the continued use of the package, Eddelbuettel decided to revitalize it. New releases, using the initial name **Rcpp**, started in November 2008. These already included an improved build and distribution process, additional documentation, and new functionality—while retaining the existing interface. This constitutes the 'classic **Rcpp**' interface (not described in this article) which will be maintained for the foreseeable future.

Yet C++ coding standards continued to evolve (Meyers, 2005). In 2009, Eddelbuettel and François started a significant redesign of the code base which added numerous new features. Several of these are described below in the section on the **Rcpp** API interface, as well as in the eight vignettes included with the package. This new API is our current focus, and we intend to both extend and support it in future development of the package.

Comparison

Integration of C++ and R has been addressed by several authors; the earliest published reference is probably Bates and DebRoy (2001). An unpublished paper by Java, Gaile, and Manly (2007) expresses several ideas that are close to some of our approaches, though not yet fully fleshed out. The **Rserve** package (Urbanek, 2003, 2010) was another early approach, going back to 2002. On the server side, **Rserve** translates R data structures into a binary serialization format and uses TCP/IP for transfer. On the client side, objects are reconstructed as instances of Java or C++ classes that emulate the structure of R objects.

The packages **rcppbind** (Liang, 2008), **RAbstraction** (Armstrong, 2009a) and **RObjects** (Armstrong, 2009b) are all implemented using C++ templates. However, neither has matured to the point of a CRAN release and it is unclear how much usage these packages are seeing beyond their own authors.

CXXR (Runnalls, 2009) comes to this topic from the other side: its aim is to completely refactor R on a stronger C++ foundation. **CXXR** is therefore concerned with all aspects of the R interpreter, REPL, threading—and object interchange between R and C++ is but one part. A similar approach is discussed by Temple Lang (2009a) who suggests making low-level internals extensible by package developers in order to facilitate extending R. Another slightly different angle is offered by Temple Lang (2009b) who uses compiler output for references on the code in order to add bindings and wrappers.

Rcpp Use Cases

The core focus of **Rcpp** has always been on allowing the programmer to add C++-based functions. Here, we use 'function' in the standard mathematical sense of providing results (output) given a set of parameters or data (input). This was facilitated from the earliest releases using C++ classes for receiving various types of R objects, converting them to C++ objects and allowing the programmer to return the results to R with relative ease.

This API therefore supports two typical use cases. First, one can think of replacing existing R code with equivalent C++ code in order to reap performance gains. This case is conceptually easy as there may not be (built- or run-time) dependencies on other C or C++ libraries. It typically involves setting up data and parameters—the right-hand side components of a function call—before making the call in order to provide the result that is to be assigned to the left-hand side. Second, **Rcpp** facilitates calling functions provided by other libraries. The use resembles the first case: data and parameters are passed via **Rcpp** to a function set-up to call code from an external library.

The Rcpp API

More recently, the **Rcpp** API has been redesigned and extended, based on the usage experience of several years of **Rcpp** deployment, needs from other projects, knowledge of the internal R API, as well as current C++ design approaches. The new features in **Rcpp** were also motivated by the needs of other projects such as **RInside** (Eddelbuettel and François, 2010) for easy embedding of R in C++ applications and **RProtoBuf** (François and Eddelbuettel, 2010) that interfaces with the Protocol Buffers library.

A First Example

```
#include <Rcpp.h>

RcppExport SEXP convolve3cpp(SEXP a, SEXP b) {
    Rcpp::NumericVector xa(a);
    Rcpp::NumericVector xb(b);
    int n_xa = xa.size(), n_xb = xb.size();
    int nab = n_xa + n_xb - 1;
    Rcpp::NumericVector xab(nab);

    for (int i = 0; i < n_xa; i++)
        for (int j = 0; j < n_xb; j++)
            xab[i + j] += xa[i] * xb[j];

    return xab;
}
```

We can highlight several aspects. First, only a single header file `Rcpp.h` is needed to use the **Rcpp** API. Second, given two arguments of type `SEXP`, a third is returned. Third, both inputs are converted to C++ vector types provided by **Rcpp** (and we have more to say about these conversions below). Fourth, the usefulness of these classes can be seen when we query the vectors directly for their size—using the `size()` member function—in order to reserve a new result type of appropriate length, and with the use of the `operator[]` to extract and set individual elements of the vector. Fifth, the computation itself is straightforward embedded looping just as in the original examples in the 'Writing R Extensions' manual (R Development Core Team, 2010a). Sixth, the return conversion from the `NumericVector` to the `SEXP` type is also automatic.

We argue that this **Rcpp**-based usage is much easier to read, write and debug than the C macro-based approach supported by R itself.

Rcpp Class hierarchy

The `Rcpp::RObject` class is the basic class of the new **Rcpp** API. An instance of the `RObject` class encapsulates an R object (`SEXP`), exposes methods that are appropriate for all types of objects and transparently manages garbage collection.

The most important aspect of the `RObject` class is that it is a very thin wrapper around the `SEXP` it encapsulates. The `SEXP` is indeed the only data member of an `RObject`. The `RObject` class does not interfere with the way R manages its memory and does not perform copies of the object into a suboptimal C++ representation. Instead, it merely acts as a proxy to the object it encapsulates so that methods applied to the `RObject` instance are relayed back to the `SEXP` in terms of the standard R API.

The `RObject` class takes advantage of the explicit life cycle of C++ objects to manage exposure of the underlying R object to the garbage collector. The `RObject` effectively treats its underlying `SEXP` as a resource. The constructor of the `RObject` class takes the necessary measures to guarantee that the underlying `SEXP` is protected from the garbage collector, and the destructor assumes the responsibility to withdraw that protection.

By assuming the entire responsibility of garbage collection, **Rcpp** relieves the programmer from writing boiler plate code to manage the protection stack with `PROTECT` and `UNPROTECT` macros.

The `RObject` class defines a set of member functions applicable to any R object, regardless of its type. This ranges from querying properties of the object (`isNULL`, `isObject`, `isS4`), management of the attributes (`attributeNames`, `hasAttribute`, `attr`) to handling of slots[1] (`hasSlot`, `slot`).

[1]The member functions that deal with slots are only applicable to S4 objects; otherwise an exception is thrown.

Derived classes

Internally, an R object must have one type amongst the set of predefined types, commonly referred to as SEXP types. The 'R Internals' manual (R Development Core Team, 2010b) documents these various types. **Rcpp** associates a dedicated C++ class for most SEXP types, and therefore only exposes functionality that is relevant to the R object that it encapsulates.

For example `Rcpp::Environment` contains member functions to manage objects in the associated environment. Similarly, classes related to vectors—`IntegerVector`, `NumericVector`, `RawVector`, `LogicalVector`, `CharacterVector`, `GenericVector` (also known as `List`) and `ExpressionVector`—expose functionality to extract and set values from the vectors.

The following sub-sections present typical uses of **Rcpp** classes in comparison with the same code expressed using functions and macros of the R API.

Numeric vectors

The following code snippet is taken from Writing R extensions (R Development Core Team, 2010a). It creates a `numeric` vector of two elements and assigns some values to it.

```
SEXP ab;
PROTECT(ab = allocVector(REALSXP, 2));
REAL(ab)[0] = 123.45;
REAL(ab)[1] = 67.89;
UNPROTECT(1);
```

Although this is one of the simplest examples in Writing R extensions, it seems verbose and it is not obvious at first sight what is happening. Memory is allocated by `allocVector`; we must also supply it with the type of data (`REALSXP`) and the number of elements. Once allocated, the `ab` object must be protected from garbage collection. Lastly, the `REAL` macro returns a pointer to the beginning of the actual array; its indexing does not resemble either R or C++.

Using the `Rcpp::NumericVector` class, the code can be rewritten:

```
Rcpp::NumericVector ab(2);
ab[0] = 123.45;
ab[1] = 67.89;
```

The code contains fewer idiomatic decorations. The `NumericVector` constructor is given the number of elements the vector contains (2), which hides a call to the `allocVector` we saw previously. Also hidden is protection of the object from garbage collection, which is a behavior that `NumericVector` inherits from `RObject`. Values are assigned to the first and second elements of the vector as `NumericVector` overloads the `operator[]`.

The snippet can also be written more concisely using the `create` static member function of the `NumericVector` class:

```
Rcpp::NumericVector ab =
  Rcpp::NumericVector::create(123.45, 67.89);
```

It should be noted that although the copy constructor of the `NumericVector` class is used, it does not imply copies of the underlying array, only the `SEXP` (*i.e.* a simple pointer) is copied.

Character vectors

A second example deals with character vectors and emulates this R code

```
> c("foo", "bar")
```

Using the traditional R API, the vector can be allocated and filled as such:

```
SEXP ab;
PROTECT(ab = allocVector(STRSXP, 2));
SET_STRING_ELT( ab, 0, mkChar("foo") );
SET_STRING_ELT( ab, 1, mkChar("bar") );
UNPROTECT(1);
```

This imposes on the programmer knowledge of `PROTECT`, `UNPROTECT`, `SEXP`, `allocVector`, `SET_STRING_ELT`, and `mkChar`.

Using the `Rcpp::CharacterVector` class, we can express the same code more concisely:

```
Rcpp::CharacterVector ab(2);
ab[0] = "foo";
ab[1] = "bar";
```

R and C++ data interchange

In addition to classes, the **Rcpp** package contains two functions to perform conversion of C++ objects to R objects and back.

C++ to R : wrap

The C++ to R conversion is performed by the `Rcpp::wrap` templated function. It uses advanced template metaprogramming techniques[2] to convert a wide and extensible set of types and classes to the most appropriate type of R object. The signature of the `wrap` template is:

```
template <typename T>
SEXP wrap(const T& object);
```

[2] A discussion of template metaprogramming (Vandevoorde and Josuttis, 2003; Abrahams and Gurtovoy, 2004) is beyond the scope of this article.

The templated function takes a reference to a 'wrappable' object and converts this object into a SEXP, which is what R expects. Currently wrappable types are :

- primitive types: int, double, ... which are converted into the corresponding atomic R vectors;
- std::string objects which are converted to R atomic character vectors;
- STL containers such as std::vector<T> or std::list<T>, as long as the template parameter type T is itself wrappable;
- STL maps which use std::string for keys (e.g. std::map<std::string,T>); as long as the type T is wrappable;
- any type that implements implicit conversion to SEXP through the operator SEXP();
- any type for which the wrap template is fully specialized.

Wrappability of an object type is resolved at compile time using modern techniques of template meta programming and class traits. The Rcpp-extending vignette discusses in depth how to extend wrap to third-party types. The **RcppArmadillo** package (François, Eddelbuettel, and Bates, 2010) features several examples. The following example shows that the design allows composition:

```
RcppExport SEXP someFunction() {
  std::vector<std::map<std::string,int> > v;
  std::map<std::string, int> m1;
  std::map<std::string, int> m2;

  m1["foo"]=1; m1["bar"]=2;
  m2["foo"]=1; m2["bar"]=2; m2["baz"]=3;

  v.push_back( m1 );
  v.push_back( m2 );
  return Rcpp::wrap( v );
}
```

The code creates a list of two named vectors, equal to the result of this R statement:

```
list( c( bar = 2L, foo = 1L) ,
      c( bar = 2L, baz = 3L, foo = 1L) )
```

R to C++ : as

The reverse conversion is implemented by variations of the Rcpp::as template whose signature is:

```
template <typename T>
T as(SEXP x) throw(not_compatible);
```

It offers less flexibility and currently handles conversion of R objects into primitive types (bool, int, std::string, ...), STL vectors of primitive types (std::vector<bool>, std::vector<double>, etc ...) and arbitrary types that offer a constructor that takes a SEXP. In addition as can be fully or partially specialized to manage conversion of R data structures to third-party types as can be seen for example in the **RcppArmadillo** package which eases transfer of R matrices and vectors to the optimised data structures in the Armadillo linear algebra library (Sanderson, 2010).

Implicit use of converters

The converters offered by wrap and as provide a very useful framework to implement code logic in terms of C++ data structures and then explicitly convert data back to R.

In addition, the converters are also used implicitly in various places in the Rcpp API. Consider the following code that uses the Rcpp::Environment class to interchange data between C++ and R.

```
// access vector 'x' from global env.
Rcpp::Environment global =
    Rcpp::Environment::global_env();
std::vector<double> vx = global["x"];

// create a map<string,string>
std::map<std::string,std::string> map;
map["foo"] = "oof";
map["bar"] = "rab";

// push the STL map to R
global["y"] = map;
```

In the first part of the example, the code extracts a std::vector<double> from the global environment. In order to achieve this, the operator[] of Environment uses the proxy pattern (Meyers, 1995) to distinguish between left hand side (LHS) and right hand side (RHS) use. The output of the operator is an instance of the nested class Environment::Binding, which defines a templated implicit conversion operator that allows a Binding to be assigned to any type that Rcpp::as is able to handle.

In the second part of the example, LHS use of the Binding instance is implemented through its assignment operator, which is also templated and uses Rcpp::wrap to perform the conversion to a SEXP that can be assigned to the requested symbol in the global environment.

The same mechanism is used throughout the API. Examples include access/modification of object attributes, slots, elements of generic vectors (lists), function arguments, nodes of dotted pair lists, language calls and more.

Environment: Using the **Rcpp** API

```
Environment stats("package:stats");
Function rnorm = stats["rnorm"];
return rnorm(10, Named("sd", 100.0));
```

Environment: Using the R API

```
SEXP stats = PROTECT(
  R_FindNamespace( mkString("stats")));
SEXP rnorm = PROTECT(
  findVarInFrame( stats, install("rnorm")));
SEXP call  = PROTECT(
  LCONS( rnorm,
    CONS(ScalarInteger(10),
      CONS(ScalarReal(100.0),R_NilValue))));
SET_TAG( CDDR(call), install("sd") );
SEXP res = PROTECT(eval(call, R_GlobalEnv));
UNPROTECT(4);
return res;
```

Language: Using the **Rcpp** API

```
Language call("rnorm",10,Named("sd",100.0));
return call.eval();
```

Language: Using the R API

```
SEXP call  = PROTECT(
  LCONS( install("rnorm"),
    CONS(ScalarInteger(10),
      CONS(ScalarReal(100.0),R_NilValue))));
SET_TAG( CDDR(call), install("sd") );
SEXP res = PROTECT(eval(call, R_GlobalEnv));
UNPROTECT(2);
return res;
```

Table 1: **Rcpp** versus the R API: Four ways of calling `rnorm(10L, sd=100)` in C / C++. We have removed the `Rcpp::` prefix for readability; this corresponds to adding a directive `using namespace Rcpp;` in the code.

Function calls

The next example shows how to use **Rcpp** to emulate the R code `rnorm(10L, sd=100.0)`. As shown in Table 1, the code can be expressed in several ways in either **Rcpp** or the standard R API. The first version shows the use of the `Environment` and `Function` classes by **Rcpp**. The second version shows the use of the `Language` class, which manage calls (LANGSXP). For comparison, we also show both versions using the standard R API.

This example illustrates that the **Rcpp** API permits us to work with code that is easier to read, write and maintain. More examples are available as part of the documentation included in the **Rcpp** package, as well as among its over seven hundred and seventy unit tests.

Using code 'inline'

Extending R with compiled code also needs to address how to reliably compile, link and load the code. While using a package is preferable in the long run, it may be too involved for quick explorations. An alternative is provided by the **inline** package (Sklyar, Murdoch, Smith, Eddelbuettel, and François, 2010) which compiles, links and loads a C, C++ or Fortran function—directly from the R prompt using simple functions `cfunction` and `cxxfunction`. The latter provides an extension which works particularly well with **Rcpp** via so-called 'plugins' which provide information about additional header file and library locations.

The use of **inline** is possible as **Rcpp** can be installed and updated just like any other R package using *e.g.* the `install.packages()` function for initial installation as well as `update.packages()` for upgrades. So even though R / C++ interfacing would otherwise require source code, the **Rcpp** library is always provided ready for use as a pre-built library through the CRAN package mechanism.[3]

The library and header files provided by **Rcpp** for use by other packages are installed along with the **Rcpp** package. The `LinkingTo: Rcpp` directive in the DESCRIPTION file lets R compute the location of the header file. The **Rcpp** package provides appropriate information for the `-L` switch needed for linking via the function `Rcpp:::LdFlags()` that provide this information. It can be used by `Makevars` files of other packages, and **inline** makes use of it internally so that all of this is done behind the scenes without the need for explicitly setting compiler or linker options.

[3]This presumes a platform for which pre-built binaries are provided. **Rcpp** is available in binary form for Windows and OS X users from CRAN, and as a `.deb` package for Debian and Ubuntu users. For other systems, the **Rcpp** library is automatically built from source during installation or upgrades.

The convolution example provided above can be rewritten for use by **inline** as shown below. The function body is provided by the character variable `src`, the function header is defined by the argument `signature`—and we only need to enable `plugin="Rcpp"` to obtain a new function `fun` based on the C++ code in `src`:

```
> src <- '
+   Rcpp::NumericVector xa(a);
+   Rcpp::NumericVector xb(b);
+   int n_xa = xa.size(), n_xb = xb.size();
+
+   Rcpp::NumericVector xab(n_xa + n_xb - 1);
+   for (int i = 0; i < n_xa; i++)
+     for (int j = 0; j < n_xb; j++)
+        xab[i + j] += xa[i] * xb[j];
+   return xab;
+ '
> fun <- cxxfunction(
+     signature(a="numeric", b="numeric"),
+     src, plugin="Rcpp")
> fun( 1:3, 1:4 )
[1]  1  4 10 16 17 12
```

Using STL algorithms

The C++ Standard Template Library (STL) offers a variety of generic algorithms designed to be used on ranges of elements (Plauger, Stepanov, Lee, and Musser, 2000). A range is any sequence of objects that can be accessed through iterators or pointers. All **Rcpp** classes from the new API representing vectors (including lists) can produce ranges through their member functions `begin()` and `end()`, effectively supporting iterating over elements of an R vector.

The following code illustrates how **Rcpp** might be used to emulate a simpler[4] version of `lapply` using the `transform` algorithm from the STL.

```
> src <- '
+   Rcpp::List input(data);
+   Rcpp::Function f(fun);
+   Rcpp::List output(input.size());
+   std::transform(
+    input.begin(), input.end(),
+    output.begin(),
+    f );
+   output.names() = input.names();
+   return output;
+   '
> cpp_lapply <- cxxfunction(
+   signature(data="list", fun = "function"),
+   src, plugin="Rcpp")
```

We can use this to calculate a summary of each column of the `faithful` data set included with R.

```
> cpp_lapply( faithful, summary )
$eruptions
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.600   2.163   4.000   3.488   4.454   5.100

$waiting
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 43.0    58.0    76.0    70.9    82.0    96.0
```

[4]The version of `lapply` does not allow use of the ellipsis (`...`).

Error handling

Code that uses both R and C++ has to deal with two concurrent error handling models. **Rcpp** simplifies this and allows both systems to work together.

C++ exceptions in R

The internals of the R condition mechanism and the implementation of C++ exceptions are both based on a layer above POSIX jumps. These layers both assume total control over the call stack and should not be used together without extra precaution. **Rcpp** contains facilities to combine both systems so that C++ exceptions are caught and recycled into the R condition mechanism.

Rcpp defines the `BEGIN_RCPP` and `END_RCPP` macros that should be used to bracket code that might throw C++ exceptions.

```
RcppExport SEXP fun( SEXP x ){
BEGIN_RCPP
    int dx = Rcpp::as<int>(x);
    if( dx > 10 )
        throw std::range_error("too big");
    return Rcpp::wrap( dx * dx);
END_RCPP
}
```

The macros are simply defined to avoid code repetition. They expand to simple try/catch blocks:

```
RcppExport SEXP fun( SEXP x ){
    try{
        int dx = Rcpp::as<int>(x);
        if( dx > 10 )
            throw std::range_error("too big");
        return Rcpp::wrap( dx * dx);
    } catch( std::exception& __ex__ ){
        forward_exception_to_r( __ex__ );
    } catch(...){
        ::Rf_error( "c++ exception "
                    "(unknown reason)" );
    }
}
```

Using `BEGIN_RCPP` and `END_RCPP` — or the expanded versions — guarantees that the stack is first unwound in terms of C++ exceptions, before the

problem is converted to the standard R error management system (`Rf_error`).

The `forward_exception_to_r` function uses runtime type information to extract information about the class of the C++ exception and its message, so that dedicated handlers can be installed on the R side.

```
> f <- function(x) .Call( "fun", x )
> tryCatch( f( 12 ),
+    "std::range_error" = function(e) {
+       conditionMessage( e )
+    } )
[1] "too big"
> tryCatch( f( 12 ),
+    "std::range_error" = function(e) {
+       class( e )
+    } )
[1] "std::range_error" "C++Error"
[3] "error"            "condition"
```

R error in C++

R currently does not offer C-level mechanisms to deal with errors. To overcome this problem, **Rcpp** uses the `Rcpp::Evaluator` class to evaluate an expression in an R-level `tryCatch` block. The error, if any, that occurs while evaluating the function is then translated into an C++ exception that can be dealt with using regular C++ try/catch syntax.

Performance comparison

In this section, we present several different ways to leverage **Rcpp** to rewrite the convolution example taken from R Development Core Team (2010a).

As part of the redesign of **Rcpp**, data copy is kept to the absolute minimum: the `RObject` class and all its derived classes are just a container for a `SEXP` object. We let R perform all memory management and access data though the macros or functions offered by the standard R API.

The implementation of the `operator[]` is designed to be as efficient as possible, using both inlining and caching, but even this implementation is still less efficient than the reference C implementation described in R Development Core Team (2010a).

Rcpp follows design principles from the STL, and classes such as `NumericVector` expose iterators that can be used for sequential scans of the data. Algorithms using iterators are usually more efficient than those that operate on objects using the `operator[]`. The following version illustrate the use of the `NumericVector::iterator`.

```
#include <Rcpp.h>

RcppExport SEXP convolve4cpp(SEXP a, SEXP b){
    Rcpp::NumericVector xa(a), xb(b);
    int n_xa = xa.size(), n_xb = xb.size();
    Rcpp::NumericVector xab(n_xa + n_xb - 1);

    typedef Rcpp::NumericVector::iterator
        vec_iterator;
    vec_iterator ia = xa.begin(),
                 ib = xb.begin();
    vec_iterator iab = xab.begin();
    for (int i = 0; i < n_xa; i++)
        for (int j = 0; j < n_xb; j++)
            iab[i + j] += ia[i] * ib[j];

    return xab;
}
```

One of the focuses of recent developments of **Rcpp** is called 'Rcpp sugar', and aims to provide R-like syntax in C++. A discussion of Rcpp sugar is beyond the scope of this article, but for illustrative purposes we have included another version of the convolution algorithm based on Rcpp sugar.

```
#include <Rcpp.h>

RcppExport SEXP convolve11cpp(SEXP a, SEXP b){
    Rcpp::NumericVector xa(a), xb(b);
    int n_xa = xa.size(), n_xb = xb.size();
    Rcpp::NumericVector xab(n_xa+n_xb-1,0.0);

    Rcpp::Range r( 0, n_xb-1 );
    for (int i=0; i<n_xa; i++, r++)
        xab[ r ] += Rcpp::noNA(xa[i]) *
                    Rcpp::noNA(xb);
    return xab ;
}
```

Rcpp sugar allows manipulation of entire subsets of vectors at once, thanks to the `Range` class. Rcpp sugar uses techniques such as expression templates, lazy evaluation and loop unrolling to generate very efficient code. The `noNA` template function marks its argument to indicates that it does not contain any missing values—an assumption made implicitly by other versions—allowing sugar to compute the individual operations without having to test for missing values.

Implementation	Time in millisec	Relative to R API
R API (as benchmark)	218	
Rcpp sugar	145	0.67
`NumericVector::iterator`	217	1.00
`NumericVector::operator[]`	282	1.29
`RcppVector<double>`	683	3.13

Table 2: Performance for convolution example

We have benchmarked the various implementations by averaging over 5000 calls of each function

with `a` and `b` containing 200 elements each.[5] The timings are summarized in Table 2.

The first implementation, written in C and using the traditional R API provides our base case. It takes advantage of pointer arithmetics, therefore does not pay the price of C++'s object encapsulation or operator overloading. The slowest implementation comes from the (deprecated) classic **Rcpp** API. It is clearly behind in terms of efficiency. The difference is mainly caused by the many unnecessary copies that the older code performs.

The second-slowest solution uses the more efficient new **Rcpp** API. While already orders of magnitude faster than the preceding solution, it illustrates the price of object encapsulation and of calling an overloaded `operator[]` as opposed to using direct pointer arithmetics as in the reference case.

The next implementation uses iterators rather than indexing. Its performance is indistinguishable from the base case. This shows that use of C++ does not necessarily imply any performance penalty.

Finally, the fastest implementation uses Rcpp sugar. It performs significantly better than the base case: explicit loop unrolling provides vectorization at the C++ level which is responsible for this speedup.

On-going development

Rcpp is in very active development: Current work in the package (and in packages such as **RcppArmadillo**) focuses on further improving interoperability between R and C++. Two core themes are 'Rcpp sugar' as well as 'Rcpp modules' both of which are discussed in specific vignettes in the package.

'Rcpp sugar' brings syntactic sugar at the C++ level, including optimized binary operators and many R functions such as `ifelse`, `sapply`, `any`, ... The main technique used in Rcpp sugar is expression templates pioneered by the Blitz++ library (Veldhuizen, 1998) and since adopted by projects such as Armadillo (Sanderson, 2010). Access to most of the `d/p/q/r`-variants of the statistical distribution functions has also been added, enabling the use of expressions such as `dnorm(X, m, s)` for a numeric vector X and scalars m and s. Similarly, and continuing Table 1, the R expression `rnorm(10L, sd=100)` can now be written in C++ as `rnorm(10, 0, 100)` where C++ semantics require the second parameter to be used.

'Rcpp modules' allows programmers to expose C++ functions and classes at the R level. This offers access to C++ code from R using even less interface code than by writing accessor function. Modules are inspired by the `Boost.Python` library (Abrahams and Grosse-Kunstleve, 2003) that provides similar functionality for Python. C++ Classes exposed by Rcpp modules are shadowed by reference classes that have been introduced in R 2.12.0.

Summary

The **Rcpp** package presented here greatly simplifies integration of compiled C++ code with R.

The class hierarchy allows manipulation of R data structures in C++ using member functions and operators directly related to the type of object being used, thereby reducing the level of expertise required to master the various functions and macros offered by the internal R API. The classes assume the entire responsibility of garbage collection of objects, relieving the programmer from book-keeping operations with the protection stack and enabling him/her to focus on the underlying problem.

Data interchange between R and C++ code—performed by the `wrap` and `as` template functions—allows the programmer to write logic in terms of C++ data structures and facilitates use of modern libraries such as the Standard Template Library and its containers and algorithms. The `wrap()` and `as()` template functions are extensible by design and can be used either explicitly or implicitly throughout the API. By using only thin wrappers around `SEXP` objects and adopting C++ idioms such as iterators, the footprint of the **Rcpp** API is very lightweight, and does not incur a significant performance penalty.

The **Rcpp** API offers opportunities to dramatically reduce the complexity of code, which should improve code readability, maintainability and reuse.

Acknowledgments

Detailed comments and suggestions by an editor as well as two anonymous referees are gratefully acknowledged. We are also thankful for code contributions by Doug Bates and John Chambers, as well as for very helpful suggestions by Uwe Ligges, Brian Ripley and Simon Urbanek concerning the build systems for different platforms. Last but not least, several users provided very fruitful ideas for new or extended features via the `rcpp-devel` mailing list.

Bibliography

D. Abrahams and R. W. Grosse-Kunstleve. *Building Hybrid Systems with Boost.Python*. Boost Consulting, 2003. URL `http://www.boostpro.com/writing/bpl.pdf`.

D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond*. Addison-Wesley, Boston, 2004.

W. Armstrong. *RAbstraction: C++ abstraction for R objects*, 2009a. URL `http://github.com/armstrtw/`

[5] The code for this example is contained in the directory `inst/examples/ConvolveBenchmarks` in the **Rcpp** package.

rabstraction. Code repository last updated July 22, 2009.

W. Armstrong. *RObjects: C++ wrapper for R objects (a better implementation of RAbstraction*, 2009b. URL http://github.com/armstrtw/RObjects. Code repository last updated November 28, 2009.

D. M. Bates and S. DebRoy. C++ classes for R objects. In K. Hornik and F. Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing (DSC 2001)*, TU Vienna, Austria, 2001.

D. Eddelbuettel and R. François. *RInside: C++ classes to embed R in C++ applications*, 2010. URL http://CRAN.R-Project.org/package=RInside. R package version 0.2.3.

D. Eddelbuettel and K. Nguyen. *RQuantLib: R interface to the QuantLib library*, 2010. URL http://CRAN.R-Project.org//package=RQuantLib. R package version 0.3.4.

R. François and D. Eddelbuettel. *RProtoBuf: R Interface to the Protocol Buffers API*, 2010. URL http://CRAN.R-Project.org//package=RProtoBuf. R package version 0.2.0.

R. François, D. Eddelbuettel, and D. Bates. *RcppArmadillo: Rcpp integration for Armadillo templated linear algebra library*, 2010. URL http://CRAN.R-Project.org//package=RcppArmadillo. R package version 0.2.7.

J. J. Java, D. P. Gaile, and K. E. Manly. R/Cpp: Interface classes to simplify using R objects in C++ extensions. Unpublished manuscript, University at Buffalo, July 2007. URL http://sphhp.buffalo.edu/biostat/research/techreports/UB_Biostatistics_TR0702.pdf.

G. Liang. *rcppbind: A template library for R/C++ developers*, 2008. URL http://R-Forge.R-Project.org//projects/rcppbind. R package version 1.0.

S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 020163371X.

S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, third edition, 2005. ISBN 978-0321334879.

P. Plauger, A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library*. Prentice Hall PTR, 2000. ISBN 978-0134376332.

R Development Core Team. *Writing R extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2010a. URL http://CRAN.R-Project.org/doc/manuals/R-exts.html.

R Development Core Team. *R internals*. R Foundation for Statistical Computing, Vienna, Austria, 2010b. URL http://CRAN.R-Project.org/doc/manuals/R-ints.html.

A. Runnalls. Aspects of CXXR internals. In *Directions in Statistical Computing*, University of Copenhagen, Denmark, 2009.

C. Sanderson. Armadillo: An open source C++ algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010. URL http://arma.sf.net.

O. Sklyar, D. Murdoch, M. Smith, D. Eddelbuettel, and R. François. *inline: Inline C, C++, Fortran function calls from R*, 2010. URL http://CRAN.R-Project.org//package=inline. R package version 0.3.6.

D. Temple Lang. A modest proposal: an approach to making the internal R system extensible. *Computational Statistics*, 24(2):271–281, May 2009a.

D. Temple Lang. Working with meta-data from C/C++ code in R: the RGCCTranslationUnit package. *Computational Statistics*, 24(2):283–293, May 2009b.

S. Urbanek. Rserve: A fast way to provide R functionality to applications. In K. Hornik, F. Leisch, and A. Zeileis, editors, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, TU Vienna, Austria, 2003.

S. Urbanek. *Rserve: Binary R server*, 2010. URL http://CRAN.R-Project.org//package=Rserve. R package version 0.6-2.

D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, Boston, 2003.

T. L. Veldhuizen. Arrays in Blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag. ISBN 3-540-65387-2.

Dirk Eddelbuettel
Debian Project
Chicago, IL
USA
edd@debian.org

Romain François
Professional R Enthusiast
1 rue du Puits du Temple, 34 000 Montpellier
FRANCE
romain@r-enthusiasts.com