Rcpp and RInside for R and C++ Integration

Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org
dirk@eddelbuettel.com

Joint work with Romain François

R/Finance 2012
Chicago, IL
11 May 2012
The three main questions for this talk:

- **Why?** There are several reasons discussed next ...
- **How?** We will show some simple illustrations ...
- **What?** This will also be covered ...
Outline

1. Why would we extend R with C++?
2. How can Rcpp help us?
3. What can we do with Rcpp?
4. What else should we know about Rcpp?
5. Who is using Rcpp?
6. RInside
Why R? – A Simple Example

Courtesy of Greg Snow via r-help during Sep 2010

```
xx <- faithful$eruptions
fit <- density(xx)
plot(fit)
```

Standard R use: load some data, estimate a density, plot it.
xx <- faithful$eruptions
fit1 <- density(xx)
fit2 <- replicate(10000, {
  x <- sample(xx, replace=TRUE);
  density(x, from=min(fit1$x),
          to=max(fit1$x))$y
})
fit3 <- apply(fit2, 1, quantile, c(0.025, 0.975))
plot(fit1, ylim=range(fit3))
polygon(c(fit1$x, rev(fit1$x)),
        c(fit3[1,], rev(fit3[2,])),
        col='grey', border=F)
lines(fit1)

What other language can do that in seven statements?
Chambers (2008) opens chapter 11 (Interfaces I: Using C and Fortran) with these words:

> Since the core of R is in fact a program written in the C language, it's not surprising that the most direct interface to non-R software is for code written in C, or directly callable from C. All the same, including additional C code is a serious step, with some added dangers and often a substantial amount of programming and debugging required. You should have a good reason.
Motivation
Why would extending R via C/C++/Rcpp be of interest?

Chambers (2008) opens chapter 11 (Interfaces I: Using C and Fortran) with these words:

Since the core of R is in fact a program written in the C language, it’s not surprising that the most direct interface to non-R software is for code written in C, or directly callable from C. All the same, including additional C code is a serious step, with some added dangers and often a substantial amount of programming and debugging required. You should have a good reason.
Motivation
Why would extending R via C/C++/Rcpp be of interest?

Chambers proceeds with this rough map of the road ahead:

Against:
- It’s more work
- Bugs will bite
- Potential platform dependency
- Less readable software

In Favor:
- New and trusted computations
- Speed
- Object references
The *why* boils down to:

- **speed!** Often a good enough reason for us ... and a major focus for us today.

- **new things!** We can bind to libraries and tools that would otherwise be unavailable.

- **references!** Chambers quote from 2008 somehow foreshadowed the work on *Reference Classes* released with R 2.12 and which work very well with **Rcpp** modules. More generally, we can do pass-by-reference in C/C++. 
Why extend with C++?

Why extend with C++?

That’s a near religious question.

- C is a plausible choice as R is written in it – but too bare.
- C++ is close to C, but “more”. Paraphrasing Meyers, we can call it a language with “four different paradigms inside”.
- C++ may be intimidating. It shouldn’t be. C++ in 2011 is very different from C++ in 1991.
- C++ is industrial strength. Many excellent libraries. Great support for scientific computing. Many APIs.
- Let’s focus on *Extending R, and taking C++ as a given*.
- Rcpp lets you extend R in the easiest possible way. C++ is just a tool in that context.
Outline

1. Why would we extend R with C++?
2. How can Rcpp help us?
3. What can we do with Rcpp?
4. What else should we know about Rcpp?
5. Who is using Rcpp?
6. RInside

Dirk Eddelbuettel
Rcpp and RInside for R and C++ Integration
Let’s recap what the “Writing R Extensions” manual says:

- The primary interface is the `.Call()` function
- It can take a variable number of `SEXP` variables on input.
- It returns a single `SEXP`.
- So *everything* revolves around `SEXP` objects.
- But ... what exactly is a `SEXP`?
The gory details are in Section 1.1 “SEXP$” of the *R Internals* manual.

SEXP$ are opaque pointers, and several distinct types are aggregated in a C union type.

Section 1.1.1 “SEXPTYPE” lists the 26 different types a SEXP could point to.

It’s a mess, but it is the best you can do if C is all you have.

There are macros systems (two unfortunately) to help shield the innards of SEXP$.

---

Dirk Eddelbuettel

Rcpp and RInside for R and C++ Integration
Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b){
  int i, n;
  double *xa, *xb, *xab; SEXP ab;
  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  n = LENGTH(a);
  PROTECT(ab = NEW_NUMERIC(n));
  xa=NUMERIC_POINTER(a);
  xb=NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  double x = 0.0, y = 0.0 ;
  for (i=0; i<n; i++) xab[i] = 0.0;
  for (i=0; i<n; i++) {
    x = xa[i]; y = xb[i];
    xab[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!
Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b){
  int i, n;
  double *xa, *xb, *xab; SEXP ab;
  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  n = LENGTH(a);
  PROTECT(ab = NEW_NUMERIC(n));
  xa=NUMERIC_POINTER(a);
  xb=NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  double x = 0.0, y = 0.0 ;
  for (i=0; i<n; i++) xab[i] = 0.0;
  for (i=0; i<n; i++) {
    x = xa[i]; y = xb[i];
    xab[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!

Or using `Rcpp`.

```c
#include <Rcpp.h>

extern "C" SEXP v2(SEXP a, SEXP b) {
  int n = x.size();
  NumericVector res(n);
  for (int i=0; i<n; i++) {
    res[i] = (x[i] < y[i]) ? x[i]*x[i] : -(y[i]*y[i]);
  }
  return res;
}
```

In R, for comparison:

```r
res <- ifelse(x < y, x*x, -y*y)
```
Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b) {
    int i, n;
    double *xa, *xb, *xab; SEXP ab;
    PROTECT(a = AS_NUMERIC(a));
    PROTECT(b = AS_NUMERIC(b));
    n = LENGTH(a);
    PROTECT(ab = NEW_NUMERIC(n));
    xa=NUMERIC_POINTER(a);
    xb=NUMERIC_POINTER(b);
    xab = NUMERIC_POINTER(ab);
    double x = 0.0, y = 0.0 ;
    for (i=0; i<n; i++) xab[i] = 0.0;
    for (i=0; i<n; i++) {
        x = xa[i]; y = xb[i];
        xab[i] = (x < y) ? x*x : -(y*y);
    }
    UNPROTECT(3);
    return (ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!

Or using Rcpp.

```c
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
    NumericVector x(a), y(b);
    int n = x.size();
    NumericVector res(n);
    for (int i=0; i<n; i++) {
        res[i] = (x[i] < y[i]) ? x[i]*x[i] : -(y[i]*y[i]);
    }
    return res;
}
```

or using Rcpp sugar:

```c
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
    NumericVector x(a), y(b);
    NumericVector res = ifelse(x < y, x*x, -(y*y));
    return res;
}
```

Dirk Eddelbuettel  
Rcpp and RInside for R and C++ Integration
Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b){
    int i, n;
    double *xa, *xb, *xab; SEXP ab;
    PROTECT(a = AS_NUMERIC(a));
    PROTECT(b = AS_NUMERIC(b));
    n = LENGTH(a);
    PROTECT(ab = NEW_NUMERIC(n));
    xa=NUMERIC_POINTER(a);
    xb=NUMERIC_POINTER(b);
    xab = NUMERIC_POINTER(ab);
    double x = 0.0, y = 0.0 ;
    for (i=0; i<n; i++) xab[i] = 0.0;
    for (i=0; i<n; i++) {
        x = xa[i]; y = xb[i];
        xab[i] = (x < y) ? x*x : -(y*y);
    }
    UNPROTECT(3);
    return (ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!

Or using Rcpp.

```cpp
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
    NumericVector x(a), y(b);
    int n = x.size();
    NumericVector res(n);
    for (int i=0; i<n; i++) {
        res[i] = (x[i] < y[i]) ? x[i]*x[i] : -(y[i]*y[i]);
    }
    return res;
}
```

or using Rcpp sugar:

```cpp
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
    NumericVector x(a), y(b);
    NumericVector res = ifelse(x < y, x*x, -(y*y));
    return res;
}
```

In R, for comparison:

```
res <- ifelse(x < y, x*x, -(y*y))
```
Comparing the R API to Rcpp: Vectors – R use
With magic provided by the 'inline' package (Sklyar et al)

R> ex1c <- cfunction(signature(a="numeric", b="numeric"),
+ body=
+ int i, n;
+ double *xa, *xb, *xab; SEXP ab;
+ PROTECT(a = AS_NUMERIC(a));
+ PROTECT(b = AS_NUMERIC(b));
+ n = LENGTH(a);
+ PROTECT(ab = NEW_NUMERIC(n));
+ xa=NUMERIC_POINTER(a);
+ xb=NUMERIC_POINTER(b);
+ xab = NUMERIC_POINTER(ab);
+ double x = 0.0, y = 0.0 ;
+ for (i=0; i<n; i++) xab[i] = 0.0;
+ for (i=0; i<n; i++) {
+ x = xa[i]; y = xb[i];
+ xab[i] = (x < y) ? x*x : -(y*y);
+ }
+ UNPROTECT(3);
+ return(ab);
+ )
R> a <- c(1,2,3,4)
R> b <- c(4,1,4,1)

R> ex1rcpp <-
+ cxxfunction(signature(a="numeric", b="numeric"),
+ plugin="Rcpp", body=
+ NumericVector x(a), y(b);
+ int n = x.size();
+ NumericVector res(n);
+ for (int i=0; i<n; i++) {
+ res[i] = (x[i] < y[i]) ?
+ x[i]*x[i] :
+ -(y[i]*y[i]);
+ }
+ return res;
+ )
R> stopifnot(all.equal(ex1c(a,b), ex1rcpp(a,b))

R> ex1rcppSugar <-
+ cxxfunction(signature(a="numeric", b="numeric"),
+ plugin="Rcpp", body=
+ NumericVector x(a), y(b);
+ NumericVector res =
+ ifelse(x < y, x*x, -(y*y));
+ return res;
+ )
R> stopifnot(all.equal(ex1c(a,b), ex1rcppSugar(a,b))
R>
Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP foobarRC()
{
    SEXP res = PROTECT(allocVector(STRSXP, 2));
    SET_STRING_ELT( res, 0, mkChar( "foo" ) ) ;
    SET_STRING_ELT( res, 1, mkChar( "bar" ) ) ;
    UNPROTECT(1) ;
    return res ;
}
```

Need to remember to use `STRSXP`, allocate vectors, set elements as string elements (different from basic vectors).

Or using Rcpp.

```cpp
#include <Rcpp.h>
extern "C" SEXP foobarRcpp()
{
    StringVector res(2);
    res[0] = "foo";
    res[1] = "bar";
    return res ;
}
```

Or using R:

```r
res <- c("foo", "bar")
```
Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP foobarRC(){
    SEXP res = PROTECT(allocVector(STRSXP, 2));
    SET_STRING_ELT( res, 0, mkChar( "foo" ) ) ;
    SET_STRING_ELT( res, 1, mkChar( "bar" ) ) ;
    UNPROTECT(1) ;
    return res ;
}
```

Need to remember to

- use `STRSXP`,
- allocate vectors,
- set elements as string elements (different from basic vectors).

Or using Rcpp.

```c
#include <Rcpp.h>
extern "C" SEXP foobarRcpp(){
    StringVector res(2);
    res[0] = "foo";
    res[1] = "bar";
    return res ;
}
```

Or using R:

```
res <- c("foo", "bar")
```

Dirk Eddelbuettel
Rcpp and RInside for R and C++ Integration
Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP foobarRC(){
    SEXP res = PROTECT(allocVector(STRSXP, 2));
    SET_STRING_ELT( res, 0, mkChar( "foo" ) ) ;
    SET_STRING_ELT( res, 1, mkChar( "bar" ) ) ;
    UNPROTECT(1) ;
    return res ;
}
```

Need to remember to

- use `STRSXP`,
- allocate vectors,
- set elements as string elements (different from basic vectors).

Or using Rcpp.

```c
#include <Rcpp.h>
extern "C" SEXP foobarRcpp(){
    StringVector res(2);
    res[0] = "foo";
    res[1] = "bar";
    return res ;
}
```
Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP foobarRC(){
    SEXP res = PROTECT(allocVector(STRSXP, 2));
    SET_STRING_ELT( res, 0, mkChar( "foo" ) ) ;
    SET_STRING_ELT( res, 1, mkChar( "bar" ) ) ;
    UNPROTECT(1) ;
    return res ;
}
```

Need to remember to
- use `STRSXP`,
- allocate vectors,
- set elements as string elements (different from basic vectors).

Or using Rcpp.

```c
#include <Rcpp.h>
extern "C" SEXP foobarRcpp(){
    StringVector res(2);
    res[0] = "foo";
    res[1] = "bar";
    return res ;
}
```

Or using R:

```r
res <- c("foo", "bar")
```
Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>

extern "C" SEXP callback(){
    SEXP call = PROTECT(LCONS(install("rnorm"),
        CONS(ScalarInteger(3),
            CONS(ScalarReal(10.0),
                CONS(ScalarReal(20.0), R_NilValue)
        )
    ));
    GetRNGstate();
    SEXP res = PROTECT(eval(call,R_GlobalEnv));
    PutRNGstate();
    UNPROTECT(2) ;
    return res ;
}
```

Or using Rcpp.

```c
#include <Rcpp.h>

extern "C" SEXP callback(){
    RNGScope s;
    Language l = Language("rnorm", 3, 10.0, 20.0);
    return l.eval(R_GlobalEnv) ;
}
```

Or using Rcpp differently.

```c
#include <Rcpp.h>

extern "C" SEXP callback(){
    RNGScope s;
    Function f = Function("rnorm");
    return f(3, 10, 20);
}
```

Or using Rcpp sugar.

```c
#include <Rcpp.h>

extern "C" SEXP callback(){
    RNGScope s;
    return rnorm(3, 10, 20);
}
```

Or using R:

```r
res <- rnorm(3, 10.0, 20.0)
```

(And essentially no timing differences.)
Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP callback(){
  SEXP call = PROTECT(LCONS(install("rnorm"),
    CONS(ScalarInteger(3),
    CONS(ScalarReal(10.0),
    CONS(ScalarReal(20.0), R_NilValue)
  ));
}
GetRNGstate();
SEXP res = PROTECT(eval(call,R_GlobalEnv));
PutRNGstate();
UNPROTECT(2);
return res;
}

Or using Rcpp.

```cpp
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Language l = Language("rnorm",
    3, 10.0, 20.0);

  return l.eval(R_GlobalEnv) ;
}
```
Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
 extern "C" SEXP callback(){
   SEXP call = PROTECT(LCONS(install("rnorm"),
       CONS(ScalarInteger(3),
           CONS(ScalarReal(10.0),
               CONS(ScalarReal(20.0), R_NilValue)
           )
       )
   );
   GetRNGstate();
   SEXP res = PROTECT(eval(call,R_GlobalEnv));
   PutRNGstate();
   UNPROTECT(2) ;
   return res ;
}
```

or using Rcpp differently

```c
#include <Rcpp.h>
 extern "C" SEXP callback(){
   RNGScope s;
   Language l = Language("rnorm",
       3, 10.0, 20.0);
   return l.eval(R_GlobalEnv) ;
}
```

Or using Rcpp.

```c
#include <Rcpp.h>
 extern "C" SEXP callback(){
   RNGScope s;
   Function f = Function("rnorm");
   return f(3, 10, 20);
}
```

or using Rcpp sugar

```c
#include <Rcpp.h>
 extern "C" SEXP callback(){
   RNGScope s;
   return rnorm(3, 10, 20);
}
```

or using R:

```r
res <- rnorm(3, 10.0, 20.0)
```

(And essentially no timing differences.)
Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>

extern "C" SEXP callback(){
    SEXP call = PROTECT(LCONS(install("rnorm"),
        CONS(ScalarInteger(3),
            CONS(ScalarReal(10.0),
                CONS(ScalarReal(20.0), R_NilValue)
            )
        )
    ));
    GetRNGstate();
    SEXP res = PROTECT(eval(call,R_GlobalEnv));
    PutRNGstate();
    UNPROTECT(2) ;
    return res ;
}
```

or using **Rcpp** differently

```cpp
#include <Rcpp.h>

extern "C" SEXP callback(){
    RNGScope s;
    Function f = Function("rnorm");
    return f(3, 10, 20);
}
```

or using **Rcpp** sugar

```cpp
#include <Rcpp.h>

extern "C" SEXP callback(){
    RNGScope s;
    return rnorm(3, 10, 20);
}
```

Or using **Rcpp**.

```cpp
#include <Rcpp.h>

extern "C" SEXP callback(){
    RNGScope s;
    Language l = Language("rnorm","rnorm",
        3, 10.0, 20.0);
    return l.eval(R_GlobalEnv) ;
}
```
Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP callback(){
    SEXP call = PROTECT(LCONS(install("rnorm"),
        CONS(ScalarInteger(3),
            CONS(ScalarReal(10.0),
                CONS(ScalarReal(20.0), R_NilValue)
        )));
    GetRNGstate();
    SEXP res = PROTECT(eval(call,R_GlobalEnv));
    PutRNGstate();
    UNPROTECT(2);
    return res;
}
```

Or using **Rcpp** differently

```c
#include <Rcpp.h>
extern "C" SEXP callback(){
    RNGScope s;
    Function f = Function("rnorm");
    return f(3, 10, 20);
}
```

Or using **Rcpp** sugar

```c
#include <Rcpp.h>
extern "C" SEXP callback(){
    RNGScope s;
    return rnorm(3, 10, 20);
}
```

or using **R**:

```r
res <- rnorm(3, 10.0, 20.0)
```

(And essentially no timing differences.)
Comparing the R API to Rcpp: Lists

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>

extern "C" SEXP listex()
{
  SEXP res = PROTECT(allocVector(VECSXP, 2));
  SEXP x1 = PROTECT(allocVector(REALSXP, 2));
  SEXP x2 = PROTECT(allocVector(INTSXP, 2));
  SEXP klass = PROTECT(mkString("foobar"));

  double* px1 = REAL(x1);
  px1[0] = 0.5;
  px1[1] = 1.5;
  int* px2 = INTEGER(x2);
  px2[0] = 2;
  px2[1] = 3;

  SET_VECTOR_ELT(res, 0, x1);
  SET_VECTOR_ELT(res, 1, x2);
  setAttrib(res, install("class"), klass);

  UNPROTECT(4);
  return res;
}
```

Or using Rcpp.

```c
#include <Rcpp.h>

extern "C" SEXP listex2()
{
  NumericVector x = NumericVector::create(.5, 1.5);
  IntegerVector y = IntegerVector::create(2, 3);
  List res = List::create(x, y);
  res.attr("class") = "foobar";
  return res;
}
```

Or using R:

```r
ex4 <- function()
{x
  y
  r <- list(x, y)
  class(r) <- "foobar"
  r
}
```

Dirk Eddelbuettel

Rcpp and RInside for R and C++ Integration
Comparing the R API to Rcpp: Lists

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>

extern "C" SEXP listex(){
    SEXP res = PROTECT(allocVector(VECSXP, 2));
    SEXP x1 = PROTECT(allocVector(REALSXP, 2));
    SEXP x2 = PROTECT(allocVector(INTSXP, 2));
    SEXP klass = PROTECT(mkString("foobar"));
    double* px1 = REAL(x1);
    px1[0] = 0.5;
    px1[1] = 1.5;
    int* px2 = INTEGER(x2);
    px2[0] = 2;
    px2[1] = 3;
    SET_VECTOR_ELT(res, 0, x1);
    SET_VECTOR_ELT(res, 1, x2);
    setAttrib(res, install("class"), klass);
    UNPROTECT(4);
    return res;
}
```

Or using Rcpp.

```c
#include <Rcpp.h>

extern "C" SEXP listex2(){
    NumericVector x=NumericVector::create(.5,1.5);
    IntegerVector y=IntegerVector::create(2, 3);
    List res =List::create(x, y);
    res.attr("class") = "foobar";
    return res;
}
```
Comparing the R API to Rcpp: Lists

Using the basic C API for R.

```c
#include <R.h>
#include <Rdefines.h>

extern "C" SEXP listex(){
  SEXP res = PROTECT(allocVector(VECSXP, 2));
  SEXP x1 = PROTECT(allocVector(REALSXP, 2));
  SEXP x2 = PROTECT(allocVector(INTSXP, 2));
  SEXP klass = PROTECT(mkString("foobar"));
  double* px1 = REAL(x1);
  px1[0] = 0.5;
  px1[1] = 1.5;
  int* px2 = INTEGER(x2);
  px2[0] = 2;
  px2[1] = 3;
  SET_VECTOR_ELT(res, 0, x1);
  SET_VECTOR_ELT(res, 1, x2);
  setAttrib(res, install("class"), klass);
  UNPROTECT(4);
  return res;
}
```

Or using Rcpp.

```c
#include <Rcpp.h>

extern "C" SEXP listex2(){
  NumericVector x=NumericVector::create(.5,1.5);
  IntegerVector y=IntegerVector::create(2, 3);
  List res =List::create(x, y);
  res.attr("class") = "foobar";
  return res;
}
```

Or using R:

```r
ex4 <- function () {
  x <- c(0.5, 1.5)
  y <- c(2L, 3L)
  r <- list(x, y)
  class(r) <- "foobar"
  r
}
```
Lists are extremely useful for parameter passing
From the RcppExamples package

#include <Rcpp.h>

RcppExport SEXP newRcppParamsExample(SEXP params) {
    try {
        // or use BEGIN_RCPP macro
        Rcpp::List rparam(params);
        // Get parameters in params.
        std::string method = Rcpp::as< std::string > (rparam["method"]);
        double tolerance = Rcpp::as< double > (rparam["tolerance"]);
        int maxIter = Rcpp::as< int > (rparam["maxIter"]);
        Rcpp::Date startDate = Rcpp::Date (rparam["startDate"]);

        Rprintf("\nIn C++, seeing the following value\n");
        Rprintf("Method argument : %s\n", method.c_str());
        Rprintf("Tolerance argument : %f\n", tolerance);
        Rprintf("MaxIter argument : %d\n", maxIter);
        Rprintf("Start date argument: %04d-%02d-%02d\n",
          startDate.getYear(), startDate.getMonth(), startDate.getDay());

        return Rcpp::List::create(Rcpp::Named("method", method),
                                   Rcpp::Named("tolerance", tolerance),
                                   Rcpp::Named("maxIter", maxIter),
                                   Rcpp::Named("startDate", startDate),
                                   Rcpp::Named("params", params)); // or use rparam
    }
    catch ( std::exception &ex ) {
        forward_exception_to_r( ex );
    }
    catch ( ...) {
        ::Rf_error( "c++ exception (unknown reason)" );
    }
    return R_NilValue; // -Wall
}
Rcpp plays well with STL algorithms and iterators

A rather convenient feature – as can be seen in this simple `lapply()` variant:

```r
R> code <- ' + Rcpp::List input(data);
+ Rcpp::Function f(fun);
+ Rcpp::List output(input.size());
+ std::transform(input.begin(), input.end(), output.begin(), f);
+ return output;
+
R> fun <- cxxfunction(signature(data="any", fun="function"),
+                    body=code, plugin="Rcpp")
R>
R> unlist( fun(1:5, sqrt) )
[1] 1.00000 1.41421 1.73205 2.00000 2.23607
R> unlist( fun(1:5, log) )
[1] 0.00000 0.693147 1.098612 1.386294 1.609438
R> unlist( fun(1:5, function(x) { sqrt(x) + log(x) } ) )
R>
```
Outline

1. Why would we extend R with C++?
2. How can Rcpp help us?
3. What can we do with Rcpp?
4. What else should we know about Rcpp?
5. Who is using Rcpp?
6. RInside
Recall that we said the *why* boiled down to speed (which we will focus on), new things and object references.

We will look at a few examples which (re-)introduce **Rcpp** concepts and extensions, and demonstrate the gains that can be had:

- Recursive functions
- Data generation requiring a loop
- A Markov Chain Monte Carlo example
- The OLS horse race
Rcpp essentials in one page

The earlier examples showed that **Rcpp**

- can both receive entire R objects: vectors, matrices, list, ... as well as basic C++ types int, double, string, ...
- can create and return R objects easily: vectors, list, functions, matrices, ...
- this makes interfacing C++ code from R so much easier
- the **inline** package facilitates prototyping

What we haven’t shown (but is extensively documented):

- how to extend **Rcpp** to wrap around other class libraries: **RcppArmadillo**, **RcppEigen**, **RcppGSL**, ...
- how to use **Rcpp** in your own packages.
Computing the Fibonacci sequence faster

A question on the StackOverflow site lead to a short blog post, and an example now included with Rcpp. The R function

```r
fibR <- function(x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return (fibR(x - 1) + fibR(x - 2));
}
```

can be replaced with this Rcpp/inline construct:

```r
incltxt <- '
int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return (fibonacci(x - 1)) + fibonacci(x - 2);
}''

fibRcpp <- cxxfunction(signature(xs="int"),
  plugin="Rcpp",
  incl=incltxt,
  body=''
  int x = Rcpp::as<int>(xs);
  return Rcpp::wrap( fibonacci(x) );
  '')
```
Running the `examples/Misc/fibonacci.r` example in the **Rcpp** package:

```r
edd@max:~$ r svn/rcpp/pkg/Rcpp/inst/examples/Misc/fibonacci.r
Loading required package: inline
Loading required package: methods
Loading required package: compiler
test replications elapsed relative user.self sys.self
3 fibRcpp(N) 1 0.095 1.0000 0.09 0.00
1 fibR(N) 1 65.813 692.7684 65.73 0.04
2 fibRC(N) 1 65.928 693.9789 65.89 0.00
edd@max:~$
```

95 milliseconds for **Rcpp**, versus 65.8 and 65.9 seconds for R and byte-compiled R — a 690-fold gain.

(Of course, even better gains come from switching to an iterative algorithm using memoization.)
Lance Bachmeier shared an example from his graduate econometrics class which we worked into an example in RcppArmadillo as well as a short blog post.

```r
## parameter and error terms used throughout
a <- matrix(c(0.5, 0.1, 0.1, 0.5), nrow=2)
e <- matrix(rnorm(10000), ncol=2)

## Let's start with the R version
rSim <- function(coeff, err) {
  simd <- matrix(0, nrow(err), ncol(err))
  for (r in 2:nrow(err)) {
    simd[r,] = coeff %*% simd[r-1,] + err[r,]
  }
  return(simd)
}

rData <- rSim(a, e)  # generated by R
```
Simulating Vector Auto Regression (VAR): C++

---

## Now load ‘inline’ to compile C++ code on the fly

```r
suppressMessages(require(inline))
code <- '
arma::mat coeff = Rcpp::as<arma::mat>(a);
arma::mat errors = Rcpp::as<arma::mat>(e);
int m = errors.n_rows; int n = errors.n_cols;
arma::mat simdata(m,n);
simdata.row(0) = arma::zeros<arma::mat>(1,n);
for (int row=1; row < m; row++) {
  simdata.row(row) = simdata.row(row-1) * arma::trans(coeff) + errors.row(row);
}
return Rcpp::wrap(simdata);
'

## create the compiled function

cppSim <- cxxfunction(signature(a="numeric", e="numeric"),
  code, plugin="RcppArmadillo")

cppData <- cppSim(a, e) # generated by C++ code

stopifnot(all.equal(rData, cppData)) # checking results
Simulating Vector Auto Regression (VAR): Result

We run the example from the **RcppArmadillo** sources:

```
edd@max:～$ r svn/rcpp/pkg/RcppArmadillo/inst/examples/varSimulation.r
test  replications elapsed  relative user.self sys.self
1   rcppSim(a, e) 100  0.032  1.000000     0.02    0.01
3  compRsim(a, e) 100 2.113  66.03125  2.090909  0.01
2    rSim(a, e) 100 4.622 144.43750  4.634634  0.00
edd@max:～$
```

**Rcpp** provides a 140-fold gain over uncompiled R; the byte compiler (new with R 2.13.0) helps by roughly halving the computation time yet is still beat by a factor of over sixty by the C++ code.
Sanjog Misra pointed me to an example by Darren Wilkinson (comparing MCMC implementations in a few languages) and a first implementation which we reworked into what became another Rcpp example (see directory GibbsCode).

Here, the bivariate distribution

\[ f(x, y) = k \cdot x^2 \cdot e^{-xy^2 - y^2 + 2y - 4x} \]

is sampled via two conditional distributions:

\[
\begin{align*}
  f(x|y) &= x^2 e^{-x(4+y^2)} & \text{\textit{\small Gamma}} \\
  f(y|x) &= e^{-0.5 \cdot 2(x+1) \cdot (y^2 - 2y/(x+1))} & \text{\textit{\small Gaussian}}
\end{align*}
\]

which cannot be vectorised due to interdependence.
The R version is pretty straightforward:

```r
## Here is the actual Gibbs Sampler
## This is Darren Wilkinson's R code (with the corrected variance)
## But we are returning only his columns 2 and 3 as the 1:N sequence
## is never used below
Rgibbs <- function(N, thin) {
  mat <- matrix(0, ncol = 2, nrow = N)
  x <- 0
  y <- 0
  for (i in 1:N) {
    for (j in 1:thin) {
      x <- rgamma(1, 3, y*y+4)
      y <- rnorm(1, 1/(x+1), 1/sqrt(2*(x+1)))
    }
    mat[, i] <- c(x, y)
  }
  mat
}
```

as is the byte-compiled variant:

```r
## We can also try the R compiler on this R function
RCgibbs <- cmpfun(Rgibbs)
```
## Now for the Rcpp version -- Notice how easy it is to code up!

```r
# Compiling and Load
RcppGibbs <- cxxfunction(signature(n="int", thin = "int"),
gibbscode, plugin="Rcpp")
```

```c++
using namespace Rcpp;
// inline does that for us already
// n and thin are SEXPs which the Rcpp::as function maps to C++ vars
int N  = as<int>(n);
int thn = as<int>(thin);
int i,j;
NumericMatrix mat(N, 2);

RNGScope scope;                  // Initialize Random number generator

// The rest of the code follows the R version
double x=0, y=0;
for (i=0; i<N; i++) {
    for (j=0; j<thn; j++) {
        x = ::Rf_rgamma(3.0,1.0/(y*y+4));
        y = ::Rf_rnorm(1.0/(x+1),1.0/sqrt(2*x+2));
    }
    mat(i,0) = x;
    mat(i,1) = y;
}
return mat;                     // Return to R
```

# Compile and Load
RcppGibbs <- cxxfunction(signature(n="int", thin = "int"),
gibbscode, plugin="Rcpp")
The results are again quite favourable to **Rcpp**, beating even the byte-compiled variant by a factor of 24:

```r
R> ## use rbenchmark package
R> N <- 10000
R> thn <- 100
R> res <- benchmark(Rgibbs(N, thn),
+                   RCgibbs(N, thn),
+                   RcppGibbs(N, thn),
+                   columns=c("test", "replications", "elapsed",
+                               "relative", "user.self", "sys.self"),
+                   order="relative",
+                   replications=10)
R> print(res)

test  replications elapsed relative user.self sys.self
3  RcppGibbs(N, thn)     10    2.972   1.0000     2.97       0
2   RCgibbs(N, thn)     10   72.919  24.5353   72.83       0
1    Rgibbs(N, thn)     10 104.830  35.2725  104.72       0

R>

NB: Not shown are numbers from a GSL version which is even faster due to a much faster Gamma distribution RNG in the GSL.
This is a recurrent theme for me going back to a question by Ivo Welch many years ago: how does one do \texttt{lm()} faster when one also wants standard errors (to simulate test size / power trade-offs) ?

I had written first versions using the first-generation, more basic \texttt{Rcpp} against the GSL, then with Armadillo, later \texttt{RcppArmadillo} and now Eigen / \texttt{RcppEigen}.

There is an older example in the \texttt{Rcpp} package which predates the add-on packages \texttt{RcppGSL} and \texttt{RcppArmadillo} – both of which implement faster \texttt{fastLm()} functions.

But the state-of-the-art variant is in the vignette of the \texttt{RcppEigen} package and part of a paper Doug Bates and I just submitted.
Faster linear regressions: Old Comparison
These implementation predate the RcppArmadillo and RcppGSL packages

Using the ancient Longley dataset:

edd@max:~/.svn/rcpp/pkg/Rcpp/inst/examples/FastLM$ ./benchmarkLongley.r
For Longley

<table>
<thead>
<tr>
<th></th>
<th>lm</th>
<th>lm.fit</th>
<th>lmGSL</th>
<th>lmArma</th>
</tr>
</thead>
<tbody>
<tr>
<td>results</td>
<td>0.00166667</td>
<td>1.488889e-04</td>
<td>2.555556e-05</td>
<td>5.222222e-05</td>
</tr>
<tr>
<td>ratios</td>
<td>1.0000000000</td>
<td>1.19403e+01</td>
<td>6.521739e+01</td>
<td>3.191489e+01</td>
</tr>
</tbody>
</table>

Using simulated data:

edd@max:~/.svn/rcpp/pkg/Rcpp/inst/examples/FastLM$ ./benchmark.r
For n=25000 and k=9

<table>
<thead>
<tr>
<th></th>
<th>lm</th>
<th>lm.fit</th>
<th>lmGSL</th>
<th>lmArma</th>
</tr>
</thead>
<tbody>
<tr>
<td>results</td>
<td>0.1669111</td>
<td>0.01412222</td>
<td>0.03103333</td>
<td>0.009722222</td>
</tr>
<tr>
<td>ratios</td>
<td>1.0000000000</td>
<td>11.81904013</td>
<td>5.37844612</td>
<td>17.168000000</td>
</tr>
</tbody>
</table>

edd@max:~/.svn/rcpp/pkg/Rcpp/inst/examples/FastLM$
## Faster linear regressions: Recent Comparison


<table>
<thead>
<tr>
<th>Method</th>
<th>Relative</th>
<th>Elapsed</th>
<th>User</th>
<th>Sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDLt</td>
<td>1.00</td>
<td>1.18</td>
<td>1.17</td>
<td>0.00</td>
</tr>
<tr>
<td>LLt</td>
<td>1.01</td>
<td>1.19</td>
<td>1.17</td>
<td>0.00</td>
</tr>
<tr>
<td>SymmEig</td>
<td>2.76</td>
<td>3.25</td>
<td>2.70</td>
<td>0.52</td>
</tr>
<tr>
<td>QR</td>
<td>6.35</td>
<td>7.47</td>
<td>6.93</td>
<td>0.53</td>
</tr>
<tr>
<td>arma</td>
<td>6.60</td>
<td>7.76</td>
<td>25.69</td>
<td>4.47</td>
</tr>
<tr>
<td>PivQR</td>
<td>7.15</td>
<td>8.41</td>
<td>7.78</td>
<td>0.62</td>
</tr>
<tr>
<td>lm.fit</td>
<td>11.68</td>
<td>13.74</td>
<td>21.56</td>
<td>16.79</td>
</tr>
<tr>
<td>GESDD</td>
<td>12.58</td>
<td>14.79</td>
<td>44.01</td>
<td>10.96</td>
</tr>
<tr>
<td>SVD</td>
<td>44.48</td>
<td>52.30</td>
<td>51.38</td>
<td>0.80</td>
</tr>
<tr>
<td>GSL</td>
<td>150.46</td>
<td>176.95</td>
<td>210.52</td>
<td>149.86</td>
</tr>
</tbody>
</table>

**Table:** `lmBenchmark` (from the **RcppEigen** package) results on a desktop computer for the default size, 100,000 × 40, full-rank model matrix running 20 repetitions for each method. Times (Elapsed, User and Sys) are in seconds.
Outline

1. Why would we extend R with C++?
2. How can Rcpp help us?
3. What can we do with Rcpp?
4. What else should we know about Rcpp?
5. Who is using Rcpp?
6. RInside
Rcpp Sugar: vectorised C++ expressions

Rcpp sugar brings *syntactic sugar* to C++ / Rcpp programming:

- vectorized expression similar to R: `ifelse(...)`
- all the standard binary and arithmetic operators
- functions such as `any()`, `all()`, `seq_along()`, `pmin()`, `pmax()`, ... and even `sapply()` and `lapply()`
- mathematic functions: `abs()`, `exp()`, `log()`, ...
- statistical d/q/p/r functions on beta, binom, cauchy, chisq, exp, f, gamma, ... distributions

Details are in the twelve-page vignette “Rcpp-sugar”.

Dirk Eddelbuettel
Rcpp and RInside for R and C++ Integration
Rcpp Modules are inspired by the Boost.Python C++ library. Some of their key features allow us

- expose functions just by declaring the interface
- expose classes similarly just via declarations
- this includes support for constructors, private and public fields, read-only as well as read-write access and more.

The “Rcpp-modules” vignette has details, and shows how to deploy Modules in your own package.
**Rcpp** provides a function `Rcpp.package.skeleton()` which extends the base R functions after which it is modeled. It creates

- basic package directory structure
- necessary files such as `src/Makevars` and `src/Makevars.win`, `NAMESPACE` and more
- a set C++ function files (header and sources), and an R function to call it
- simple documentation files

The vignette “Rcpp-package” discusses this in more detail.
Outline

1. Why would we extend R with C++?
2. How can Rcpp help us?
3. What can we do with Rcpp?
4. What else should we know about Rcpp?
5. Who is using Rcpp?
6. RInside
CRAN Packages using Rcpp
As of early May 2012, these 66 packages use Rcpp

acer, apcluster, auteur, bcp, bfa, bifactorial, cda, fastGHQuad, fdaMixed, forecast, growcurves, GUTS, highlight, KernSmoothIRT, LaF, maxent, minqa, mirt, multmod, mvabund, NetworkAnalysis, nfda, openair, orQA, parser, phom, phylobase, planar, psgp, Rclusterpp, RcppArmadillo, RcppBDT, rcppbugs, RcppClassic, RcppDE, RcppEigen, RcppExamples, RcppGSL, RcppSMC, rgam, RInside, Rmalschains, Rmixmod, robustHD, rococo, RProtoBuf, RQuantLib, RSNNS, RSofia, rugarch, RVowpalWabbit, SBSA, sdcMicro, sdcTable, simFrame, spacodiR, sparseLTSEigen, SpatialTools, survSNP, termstrc, unmarked, VIM, waffle, WideLM, wordcloud, zic
CRAN Packages using Rcpp

We can identify some broad categories among these packages:

- packages which re-implement already existing R code in C++ for greater speed: `bcp`, `termstr`, `wordcloud`
- packages which connect to external libraries: `RQuantLib`, `RProtoBuf`, `RSNNS`, `RSofia`, `RVowpalWabbit`
- packages directly related to Rcpp providing glue to other libraries: `RcppArmadillo`, `RcppEigen`, `RcppGSL`
- packages using Rcpp Modules to easily interface C++ code: `RcppBDT`, `cds`, `planar`
Outline

1. Why would we extend R with C++?
2. How can Rcpp help us?
3. What can we do with Rcpp?
4. What else should we know about Rcpp?
5. Who is using Rcpp?
6. RInside

RInside makes it trivial to embed R
This is rinside_sample12.cpp from the RInside examples

```cpp
#include <RInside.h> // for the embedded R via RInside

int main(int argc, char *argv[]) {
    RInside R(argc, argv); // create an embedded R instance

    std::string cmd = "set.seed(123); sample(LETTERS[1:5], 10, replace=TRUE)";
    Rcpp::CharacterVector res = R.parseEval(cmd); // parse, eval + return result

    for (int i=0; i<res.size(); i++) { // loop over vector and output
        std::cout << res[i];
    }
    std::cout << std::endl;

    std::copy(res.begin(), res.end(), // or use STL iterators
               std::ostream_iterator<char*>(std::cout));
    std::cout << std::endl;

    exit(0);
}
```
RInside makes it trivial to run R examples
This is rinside_sample4.cpp from the RInside examples (minus try/catch)

```
#include <RInside.h>
#include <iomanip>

int main(int argc, char *argv[]) {
    RInside R(argc, argv); // create an embedded R instance
    std::string txt =
        "suppressMessages(library(fPortfolio)); lppData <- 100 * LPP2005.RET[, 1:6];"
        "ewSpec <- portfolioSpec(); nAssets <- ncol(lppData);"
    R.parseEvalQ(txt); // prepare problem
    const double dvec[6] = { 0.1, 0.1, 0.1, 0.1, 0.3, 0.3 }; // choose any weights
    const std::vector<double> w(dvec, &dvec[6]);
    R["weightsvec"] = w; // assign weights
    txt = "setWeights(ewSpec) <- weightsvec";
    R.parseEvalQ(txt); // evaluate assignment
    txt = "ewPf <- feasiblePortfolio(data=lppData, spec=ewSpec,"
          " constraints="LongOnly")";
    " print(ewPf); vec <- getCovRiskBudgets(ewPf@portfolio)"
    Rcpp::NumericVector V( (SEXP) R.parseEval(txt) );
    Rcpp::CharacterVector names( (SEXP) R.parseEval("names(vec)") );

    std::cout << "\n\nAnd now from C++\n\n";
    for (int i=0; i<names.size(); i++) {
        std::cout << std::setw(16) << names[i] << "\t"
                   << std::setw(11) << V[i] << "\n";
    }
    exit(0);
}
```
Building applications with RInside

This looks scarier than it really is

- We need to compile and link against R, Rcpp and RInside.
- As we can assume that R is present, we can evaluate snippets passed from the Makefile to Rscript to get an autoconfiguration scheme.
- See the Makefile in examples/standard: just drop another example file mytest.cpp and the mytest application will be built upon running make.
- Idem on Windows using Makefile.win.
- Plus, we now have contributed cmake configuration useable from Eclipse, KDevelop and Code::Blocks.
RInside allows us to embed R in desktop applications. This uses the Qt C++ toolkit (cf examples/qt in RInside).

This example is discussed more fully on my blog, and the full sources are included in the RInside package.
RInside also allows us to embed R in web applications. This uses the Wt C++ toolkit (cf examples/wt in RInside).

This example is now included with the RInside release.
... and even a dressier one with CSS and XML

Overview

This example demonstrates some of the capabilities of the Qt library, in combination with the RInside classes for embedding the R statistical language and environment.

It reimplements a standard GUI / application setting: drawing from a random distribution, and estimation a non-parametric density for which the user selects the kernel and bandwidth. RInside already contains an example of this using Qt to provide a standard application.

Here we show how to do the same in a web application which, thanks to the abstractions provided by the Qt, is rather straightforward.

User Input for Density Estimation

Density estimation scale factor (div. by 100)
100
R Command for data generation

Resulting R Chart
That’s it for today

For more information:

- the eight pdf vignettes in the **Rcpp** package (which includes our *Journal of Statistical Software* paper)
- Dirk’s site, code section and blog: [http://dirk.eddelbuettel.com](http://dirk.eddelbuettel.com)
- CRAN page(s): [http://cran.r-project.org/web/packages/Rcpp/index.html](http://cran.r-project.org/web/packages/Rcpp/index.html)
- The rcpp-devel mailing list.