Rcpp Masterclass / Workshop
Part II: Rcpp Details

Dirk Eddelbuettel[1] Romain François[2]

[1] Debian Project

[2] R Enthusiasts

28 April 2011
preceding *R / Finance 2011*
University of Illinois at Chicago

Outline

1. Main Rcpp Classes
   - RObject
   - IntegerVector
   - NumericVector
   - GenericVector
   - DataFrame
   - Function
   - Environments
   - S4

RObject

The `RObject` class is the basic class behind the new API.

It is a thin wrapper around a `SEXP` object—this is often called a *proxy model* as we do not copy the R object.

`RObject` manages the life cycle, the object is protected from garbage collection while in scope—so *you* do not have to do memory management.

`RObject` defines several member functions common to all objects (*e.g.*, `isS4()`, `attributeNames`, ...); derived classes then define specific member functions.

Overview of classes: Comparison

Rcpp class	R `typeof`
`Integer(Vector|Matrix)`	`integer` vectors and matrices
`Numeric(Vector|Matrix)`	`numeric` ...
`Logical(Vector|Matrix)`	`logical` ...
`Character(Vector|Matrix)`	`character` ...
`Raw(Vector|Matrix)`	`raw` ...
`Complex(Vector|Matrix)`	`complex` ...
`List`	`list` (aka generic vectors) ...
`Expression(Vector|Matrix)`	`expression` ...
`Environment`	`environment`
`Function`	`function`
`XPtr`	`externalptr`
`Language`	`language`
`S4`	`S4`
...	...

Overview of key vector / matrix classes

- `IntegerVector` vectors of type `integer`
- `NumericVector` vectors of type `numeric`
- `RawVector` vectors of type `raw`
- `LogicalVector` vectors of type `logical`
- `CharacterVector` vectors of type `character`
- `GenericVector` generic vectors implementing `list` types

Common core functions for Vectors and Matrices

Key operations for all vectors, styled after STL vectors:

`operator()` access elements via `()`

`operator[]` access elements via `[]`

`length()` also aliased to `size()`

`fill(u)` fills vector with value of `u`

`begin()` pointer to beginning of vector, for iterators

`end()` pointer to one past end of vector

`push_back(x)` insert x at end, grows vector

`push_front(x)` insert x at beginning, grows vector

`insert(i, x)` insert x at position i, grows vector

`erase(i)` remove element at position i, shrinks vector

Outline

1. Main Rcpp Classes
   - RObject
   - IntegerVector
   - NumericVector
   - GenericVector
   - DataFrame
   - Function
   - Environments
   - S4

A first example
examples/part2/intVecEx1.R

Let us reimplement (a simpler version of) `prod()` for integer vectors:

```
library(inline)

src <- '
    Rcpp::IntegerVector vec(vx);
    int prod = 1;
    for (int i=0; i<vec.size(); i++) {
        prod *= vec[i];
    }
    return Rcpp::wrap(prod);
'
fun <- cxxfunction(signature(vx="integer"),
                   src, plugin="Rcpp")
fun(1L:10L)
```

Passing data from from R

examples/part2/intVecEx1.R

We instantiate the IntegerVector object with the SEXP received from R:

```
library(inline)

src <- '
    Rcpp::IntegerVector vec(vx);
    int prod = 1;
    for (int i=0; i<vec.size(); i++) {
        prod *= vec[i];
    }
    return Rcpp::wrap(prod);
'
fun <- cxxfunction(signature(vx="integer"),
                   src, plugin="Rcpp")
fun(1L:10L)
```

Objects tell us their size

examples/part2/intVecEx1.R

The loop counter can use the information from the `IntegerVector` itself:

```
library(inline)

src <- '
    Rcpp::IntegerVector vec(vx);
    int prod = 1;
    for (int i=0; i<vec.size(); i++) {
        prod *= vec[i];
    }
    return Rcpp::wrap(prod);
'
fun <- cxxfunction(signature(vx="integer"),
                   src, plugin="Rcpp")
fun(1L:10L)
```

Element access

examples/part2/intVecEx1.R

We simply access elements by index (but note that the range is over $0 \ldots N-1$ as is standard for C and C++):

```
library(inline)

src <- '
    Rcpp::IntegerVector vec(vx);
    int prod = 1;
    for (int i=0; i<vec.size(); i++) {
        prod *= vec[i];
    }
    return Rcpp::wrap(prod);
'
fun <- cxxfunction(signature(vx="integer"),
                   src, plugin="Rcpp")
fun(1L:10L)
```

Returning results

examples/part2/intVecEx1.R

We return the scalar `int` by using the `wrap` helper:

```
library(inline)

src <- '
    Rcpp::IntegerVector vec(vx);
    int prod = 1;
    for (int i=0; i<vec.size(); i++) {
        prod *= vec[i];
    }
    return Rcpp::wrap(prod);
'
fun <- cxxfunction(signature(vx="integer"),
                   src, plugin="Rcpp")
fun(1L:10L)
```

An STL variant

examples/part2/intVecEx2.R

As an alternative, the Standard Template Library also allows us a loop-less variant similar in spirit to vectorised R expressions:

```
library(inline)

src <- '
  Rcpp::IntegerVector vec(vx);
  int prod = std::accumulate(vec.begin(), vec.end(),
                             1, std::multiplies<int>());
  return Rcpp::wrap(prod);
'
fun <- cxxfunction(signature(vx="integer"),
                   src, plugin="Rcpp")
fun(1L:10L)
```

Outline

1. Main Rcpp Classes
   - RObject
   - IntegerVector
   - NumericVector
   - GenericVector
   - DataFrame
   - Function
   - Environments
   - S4

A first example

examples/part2/numVecEx1.R

NumericVector is very similar to IntegerVector.

Here is an example generalizing sum of squares by supplying an exponentiation argument:

```
src <- '
  Rcpp::NumericVector vec(vx);
  double p = Rcpp::as<double>(dd);
  double sum = 0.0;
  for (int i=0; i<vec.size(); i++) {
    sum += pow(vec[i], p);
  }
  return Rcpp::wrap(sum); '
fun <- cxxfunction(signature(vx="numeric",
                             dd="numeric"),
                   src, plugin="Rcpp")
fun(1:4,2)
fun(1:4,2.2)
```

A second example

Remember to clone: examples/part2/numVecEx2.R

```
R> src <- '
+    NumericVector x1(xs);
+    NumericVector x2(Rcpp::clone(xs));
+    x1[0] = 22;
+    x2[1] = 44;
+    return(DataFrame::create(Named("orig", xs),
+                             Named("x1", x1),
+                             Named("x2", x2)));'
R> fun <- cxxfunction(signature(xs="numeric"),
+                    body=src, plugin="Rcpp")
R> fun(seq(1.0, 3.0, by=1.0))
  orig x1 x2
1   22 22  1
2    2  2 44
3    3  3  3
R>
```

A second example: continued

So why is the second case different? `examples/part2/numVecEx2.R`

Understanding why these two examples perform differently is important:

```
R> fun(seq(1.0, 3.0, by=1.0))
  orig x1 x2
1   22 22  1
2    2  2 44
3    3  3  3
R> fun(1L:3L)
  orig x1 x2
1    1 22  1
2    2  2 44
3    3  3  3
R>
```

Constructor overview

For NumericVector and other vectors deriving from RObject

```
SEXP x;
NumericVector y( x ); // from a SEXP

// cloning (deep copy)
NumericVector z = clone<NumericVector>( y );

// of a given size (all elements set to 0.0)
NumericVector y( 10 );

// ... specifying the value
NumericVector y( 10, 2.0 );

// ... with elements generated
NumericVector y( 10, ::Rf_unif_rand );

// with given elements
NumericVector y = NumericVector::create( 1.0, 2.0 );
```

Matrices

examples/part2/numMatEx1.R

`NumericMatrix` is a specialisation using a dimension attribute:

```
src <- '
  Rcpp::NumericVector mat =
         Rcpp::clone<Rcpp::NumericMatrix>(mx);
  std::transform(mat.begin(), mat.end(),
                 mat.begin(), ::sqrt);
  return mat; '
fun <- cxxfunction(signature(mx="numeric"), src,
                   plugin="Rcpp")
orig <- matrix(1:9, 3, 3)
fun(orig)
```

Matrices: RcppArmadillo for math

examples/part2/numMatEx3.R

However, **Armadillo** is an excellent C++ choice for linear algebra, and **RcppArmadillo** makes this very easy to use:

```
src <- '
  arma::mat m1 = Rcpp::as<arma::mat>(mx);
  arma::mat m2 = m1 + m1;
  arma::mat m3 = m1 * 2;
  return Rcpp::List::create(m1, m2); '
fun <- cxxfunction(signature(mx="numeric"), src,
                   plugin="RcppArmadillo")
mat <- matrix(1:9, 3, 3)
fun(mat)
```

We will say more about **RcppArmadillo** later.

Other vector types

`LogicalVector` is very similar to `IntegerVector` as it represent the two possible values of a logical, or boolean, type. These values—True and False—can also be mapped to one and zero (or even a more general 'not zero' and zero).

The class `CharacterVector` can be used for vectors of R character vectors ("strings").

The class `RawVector` can be used for vectors of raw strings.

`Named` can be used to assign named elements in a vector, similar to the R construct `a <- c(foo=3.14, bar=42)` letting us set attribute names (example below).

Outline

1. Main Rcpp Classes
   - RObject
   - IntegerVector
   - NumericVector
   - GenericVector
   - DataFrame
   - Function
   - Environments
   - S4

GenericVector class (aka List) to receive values

We can use the `List` type to receive parameters from R. This is an example from the **RcppExamples** package:

```
RcppExport SEXP newRcppParamsExample(SEXP params) {

  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"]);
  [...]
```

A `List` is initialized from a `SEXP`; elements are looked up by name as in R.

Lists can be nested too, and may contain other `SEXP` types too.

GenericVector class (aka List) to return values

We can also use the `List` type to send results from R. This is an example from the **RcppExamples** package:

```
return Rcpp::List::create(Rcpp::Named("method", method),
                          Rcpp::Named("tolerance", tolerance),
                          Rcpp::Named("maxIter", maxIter),
                          Rcpp::Named("startDate", startDate),
                          Rcpp::Named("params", params));
```

This uses the `create` method to assemble a `List` object. We use `Named` to pair each element (which can be anything wrap'able to `SEXP`) with a name.

Outline

DataFrame class

examples/part2/dataFrameEx1.R

The DataFrame class be used to receive and return values. On input, we can extract columns from a data frame; row-wise access is not possible.

```
src <- '
  Rcpp::IntegerVector v =
                 Rcpp::IntegerVector::create(1,2,3);
  std::vector<std::string> s(3);
  s[0] = "a";
  s[1] = "b";
  s[2] = "c";
  return Rcpp::DataFrame::create(Rcpp::Named("a")=v,
                                  Rcpp::Named("b")=s);
'
fun <- cxxfunction(signature(), src, plugin="Rcpp")
fun()
```

Outline

1. Main Rcpp Classes
   - RObject
   - IntegerVector
   - NumericVector
   - GenericVector
   - DataFrame
   - Function
   - Environments
   - S4

Function: First example

examples/part2/functionEx1.R

Functions are another types of `SEXP` object we can represent:

```
src <- '
   Function sort(x) ;
   return sort( y, Named("decreasing", true));'
fun <- cxxfunction(signature(x="function",
                             y="ANY"),
                   src, plugin="Rcpp")
fun(sort, sample(1:5, 10, TRUE))
fun(sort, sample(LETTERS[1:5], 10, TRUE))
```

The R function `sort` is used to instantiate a C++ object of the same name—which we feed the second argument as well as another R expression created on the spot as `decreasing=TRUE`.

Function: Second example

examples/part2/functionEx1.R

We can use the `Function` class to access R functions:

```
src <- '
  Rcpp::Function rt("rt");
  return rt(5, 3);
'
fun <- cxxfunction(signature(),
                   src, plugin="Rcpp")
set.seed(42)
fun()
```

The R function `rt()` is access directly and used to instantiate a C++ object of the same name—which we get draw five random variable with five degrees of freedom.

Outline

1. Main Rcpp Classes
   - RObject
   - IntegerVector
   - NumericVector
   - GenericVector
   - DataFrame
   - Function
   - Environments
   - S4

Environments

examples/part2/environmentEx1.R

The `Environment` class helps us access R environments.

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

fun <- cxxfunction(signature(),
                   src, plugin="Rcpp")
fun()
```

The environement of the (base) package **stats** is instatiated, and we access the `rnorm()` function from it. This is an alternative to accessing build-in functions.

Outline

1. Main Rcpp Classes
   - RObject
   - IntegerVector
   - NumericVector
   - GenericVector
   - DataFrame
   - Function
   - Environments
   - S4

S4

examples/part2/S4ex1.R

S4 classes can also be created, or altered, at the C++ level.

```
src <- '
  S4 foo(x) ;
  foo.slot(".Data") = "bar" ;
  return(foo);
'
fun <- cxxfunction(signature(x="any"), src,
                   plugin="Rcpp")
setClass( "S4ex", contains = "character",
          representation( x = "numeric" ) )
x <- new( "S4ex", "bla", x = 10 )
fun(x)
str(fun(x))
```

Outline

`as()` and `wrap()`

`as()` and `wrap()` are key components of the R and C++ data interchange.

They are declared as

```
// conversion from R to C++
template <typename T>
T as( SEXP m_sexp) throw(not_compatible);

// conversion from C++ to R
template <typename T>
SEXP wrap(const T& object);
```

as and wrap usage example

examples/part2/asAndWrapEx1.R

```
code <- '
  // we get a list from R
  Rcpp::List input(inp);
  // pull std::vector<double> from R list
  // via an implicit call to Rcpp::as
  std::vector<double> x = input["x"] ;
  // return an R list
  // via an implicit call to Rcpp::wrap
  return Rcpp::List::create(
    Rcpp::Named("front", x.front()),
    Rcpp::Named("back", x.back())
  );
'

fun <- cxxfunction(signature(inp = "list"),
                   code, plugin = "Rcpp")
input <- list(x = seq(1, 10, by = 0.5))
fun(input)
```

Outline

2. Extending Rcpp via `as` and `wrap`
   - Introduction
   - Extending wrap
   - Extending as

Extending wrap: Intrusively

We can declare a new conversion to `SEXP` operator for class `Foo` in a header `Foo.h` *before* the header `Rcpp.h` is included.

```
#include <RcppCommon.h>

class Foo {
    public:
        Foo();

        // this operator enables implicit Rcpp::wrap
        operator SEXP();
}

#include <Rcpp.h>
```

The definition can follow in a regular `Foo.cpp` file.

Extending wrap: Non-Intrusively

If we cannot modify the class of the code for which we need a wrapper, but still want automatic conversion we can use a template specialization for `wrap`:

```
#include <RcppCommon.h>

// third party library that declares class Bar
#include <foobar.h>

// declaring the specialization
namespace Rcpp {
    template <> SEXP wrap( const Bar& );
}

// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```

Extending wrap: Partial specialization

We can also declare a partial specialization as the compiler will pick the appropriate overloading:

```
#include <RcppCommon.h>

// third party library that declares template class Bling<T>
#include <foobar.h>

// declaring the partial specialization
namespace Rcpp {
    namespace traits {

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

    }
}

// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```

Outline

Extending as: Intrusively

Just like for `wrap`, we can provide an intrusive conversion by declaring a new constructor from `SEXP` for class `Foo` *before* the header `Rcpp.h` is included:

```
#include <RcppCommon.h>

class Foo{
    public:
        Foo() ;

        // this constructor enables implicit Rcpp::as
        Foo(SEXP) ;
}

#include <Rcpp.h>
```

Extending as: Non-Intrusively

We can also use a full specialization of `as` in a non-intrusive manner:

```
#include <RcppCommon.h>

// third party library that declares class Bar
#include <foobar.h>

// declaring the specialization
namespace Rcpp {
    template <> Bar as( SEXP ) throw(not_compatible) ;
}

// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```

Extending as: Partial specialization

`Rcpp::as` does not allow partial specialization. We can specialize `Rcpp::traits::Exporter`.
Partial specialization of class templayes is allowed; we can do

```
#include <RcppCommon.h>
// third party library that declares template class Bling<T>
#include <foobar.h>

// declaring the partial specialization
namespace Rcpp {
  namespace traits {
      template <typename T> class Exporter< Bling<T> >;
  }
}
// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```

Requirements for the `Exporter< Bling<T> >` class are that it should have a constructor taking a `SEXP`, and it should have a methods called `get` that returns a `Bling<T>` instance.

Outline

3 Using Rcpp in your package
- Overview
- Call
- C++ files
- R file
- DESCRIPTION and NAMESPACE
- Makevars and Makevars.win

Creating a package with Rcpp

R provides a very useful helper function to create packages: `package.skeleton()`.

We have wrapped / extended this function to `Rcpp.package.skeleton()` to create a framework for a user package.

The next few slides will show its usage.

Outline

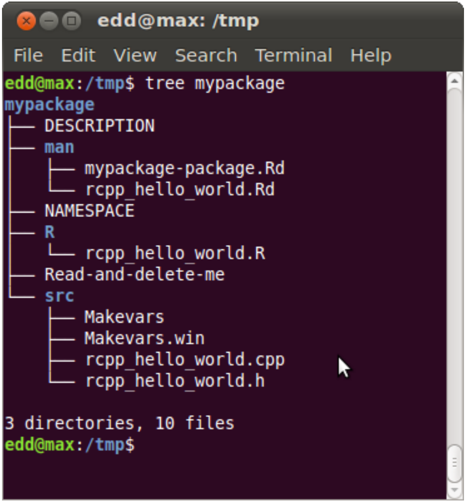
3. Using Rcpp in your package
   - Overview
   - Call
   - C++ files
   - R file
   - DESCRIPTION and NAMESPACE
   - Makevars and Makevars.win

Calling Rcpp.package.skeleton()

```
R> Rcpp.package.skeleton( "mypackage" )
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './mypackage/Read-and-delete-me'.

Adding Rcpp settings
 >> added Depends: Rcpp
 >> added LinkingTo: Rcpp
 >> added useDynLib directive to NAMESPACE
 >> added Makevars file with Rcpp settings
 >> added Makevars.win file with Rcpp settings
 >> added example header file using Rcpp classes
 >> added example src file using Rcpp classes
 >> added example R file calling the C++ example
 >> added Rd file for rcpp_hello_world
```

Rcpp.package.skeleton creates a file tree



We will discuss the individual files in the next few slides.

Outline

3. Using Rcpp in your package
   - Overview
   - Call
   - C++ files
   - R file
   - DESCRIPTION and NAMESPACE
   - Makevars and Makevars.win

The C++ header file

```
#ifndef _mypackage_RCPP_HELLO_WORLD_H
#define _mypackage_RCPP_HELLO_WORLD_H

#include <Rcpp.h>

/*
 * note : RcppExport is an alias to ‘extern "C"‘ defined by Rcpp.
 *
 * It gives C calling convention to the rcpp_hello_world function so that
 * it can be called from .Call in R. Otherwise, the C++ compiler mangles the
 * name of the function and .Call can’t find it.
 *
 * It is only useful to use RcppExport when the function is intended to be called
 * by .Call. See http://thread.gmane.org/gmane.comp.lang.r.rcpp/649/focus=672
 * on Rcpp-devel for a misuse of RcppExport
 */
RcppExport SEXP rcpp_hello_world() ;

#endif
```

The C++ source file

```
#include "rcpp_hello_world.h"

SEXP rcpp_hello_world(){
  using namespace Rcpp ;

  CharacterVector x = CharacterVector::create( "foo", "bar" )  ;
  NumericVector y   = NumericVector::create( 0.0, 1.0 ) ;
  List z            = List::create( x, y ) ;

  return z ;
}
```

Outline

3. Using Rcpp in your package
   - Overview
   - Call
   - C++ files
   - R file
   - DESCRIPTION and NAMESPACE
   - Makevars and Makevars.win

The R file

The R file makes one call to the one C++ function:

```
rcpp_hello_world <- function(){
    .Call( "rcpp_hello_world",
           PACKAGE = "mypackage" )
}
```

Outline

3. Using Rcpp in your package
   - Overview
   - Call
   - C++ files
   - R file
   - DESCRIPTION and NAMESPACE
   - Makevars and Makevars.win

The DESCRIPTION file

This declares the dependency of your package on **Rcpp**.

```
Package: mypackage
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2011-04-19
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What Licence is it under ?
LazyLoad: yes
Depends: Rcpp (>= 0.9.4)
LinkingTo: Rcpp
```

The NAMESPACE file

Here we use a regular expression to export all symbols.

```
useDynLib(mypackage)
exportPattern("^[[:alpha:]]+")
```

Outline

3 Using Rcpp in your package

- Overview
- Call
- C++ files
- R file
- DESCRIPTION and NAMESPACE
- Makevars and Makevars.win

The standard Makevars file

```
## Use the R_HOME indirection to support installations of multiple R version
PKG_LIBS = `$(R_HOME)/bin/Rscript -e "Rcpp:::LdFlags()"`

## As an alternative, one can also add this code in a file 'configure'
##
##      PKG_LIBS=`${R_HOME}/bin/Rscript -e "Rcpp:::LdFlags()"`
##
##      sed -e "s|@PKG_LIBS@|${PKG_LIBS}|" \
##          src/Makevars.in > src/Makevars
##
## which together with the following file 'src/Makevars.in'
##
##      PKG_LIBS = @PKG_LIBS@
##
## can be used to create src/Makevars dynamically. This scheme is more
## powerful and can be expanded to also check for and link with other
## libraries.  It should be complemented by a file 'cleanup'
##
##      rm src/Makevars
##
## which removes the autogenerated file src/Makevars.
##
## Of course, autoconf can also be used to write configure files. This is
## done by a number of packages, but recommended only for more advanced users
## comfortable with autoconf and its related tools.
```

The Windows `Makevars.win` file

On Windows we have to also reflect 32- and 64-bit builds in the call to `Rscript`:

```
## Use the R_HOME indirection to support installations of multiple R version
PKG_LIBS = \
   $(shell "${R_HOME}/bin${R_ARCH_BIN}/Rscript.exe" \
           -e "Rcpp:::LdFlags()")
```

Installation and Usage

```
edd@max:/tmp$ R CMD INSTALL mypackage
* installing to library '/usr/local/lib/R/site-library'
* installing *source* package 'mypackage' ...
** libs
g++ -I/usr/share/R/include [....]
g++ -shared -o mypackage.so [....]
installing to /usr/local/lib/R/site-library/mypackage/libs
** R
** preparing package for lazy loading
** help
*** installing help indices
** building package indices ...
** testing if installed package can be loaded

* DONE (mypackage)
edd@max:/tmp$ Rscript -e 'library(mypackage); rcpp_hello_world()'
Loading required package: Rcpp
Loading required package: methods
[[1]]
[1] "foo" "bar"

[[2]]
[1] 0 1

edd@max:/tmp$
```

Outline

4. Syntactic sugar
   - Motivation
   - Contents
   - Operators
   - Functions
   - Performance

Motivating Sugar

Recall the earlier example of a simple (albeit contrived for the purposes of this discussion) R vector expression:

```
ifelse(x < y, x*x, -(y*y))
```

which for a given vector `x` will execute a simple transformation.

We saw a basic C implementation. How would we write it in C++ ?

Motivating sugar

examples/part2/sugarEx1.cpp

Maybe like this.

```
SEXP foo(SEXP xx, SEXP yy) {
  int n = x.size();
  NumericVector res1( n );
  double x_ = 0.0, y_ = 0.0;
  for (int i=0; i<n; i++) {
    x_ = x[i];
    y_ = y[i];
    if (R_IsNA(x_) || R_IsNA(y_)) {
      res1[i] = NA_REAL;
    } else if (x_ < y_) {
      res1[i] = x_ * x_;
    } else {
      res1[i] = -(y_ * y_);
    }
  }
  return(x);
}
```

Motivating sugar

examples/part2/sugarEx2.cpp

But with `sugar` we can simply write it as

```
SEXP foo( SEXP xx, SEXP yy) {
  NumericVector x(xx), y(yy) ;
  return ifelse( x < y, x*x, -(y*y) ) ;
}
```

Sugar: Another example

examples/part2/sugarEx3.cpp

Sugar also gives us things like `sapply` on C++ vectors:

```
double square( double x){
  return x*x ;
}

SEXP foo( SEXP xx ){
  NumericVector x(xx) ;
  return sapply( x, square ) ;
}
```

Outline

Sugar: Overview of Contents

logical operators <, >, <=, >=, ==, !=

arithmetic operators +, -, *, /

functions on vectors `abs`, `all`, `any`, `ceiling`, `diag`, `diff`, `exp`, `head`, `ifelse`, `is_na`, `lapply`, `pmin`, `pmax`, `pow`, `rep`, `rep_each`, `rep_len`, `rev`, `sapply`, `seq_along`, `seq_len`, `sign`, `sum`, `tail`

functions on matrices `outer`, `col`, `row`, `lower_tri`, `upper_tri`, `diag`

statistical functions (dpqr) `rnorm`, `dpois`, `qlogis`, etc ...

More information in the `Rcpp-sugar` vignette.

Outline

4 Syntactic sugar
- Motivation
- Contents
- Operators
- Functions
- Performance

Binary arithmetic operators

Sugar defines the usual binary arithmetic operators : +, −, *, /.

```
// two numeric vectors of the same size
NumericVector x ;
NumericVector y ;

// expressions involving two vectors
NumericVector res = x + y ;
NumericVector res = x - y ;
NumericVector res = x * y ;
NumericVector res = x / y ;

// one vector, one single value
NumericVector res = x + 2.0 ;
NumericVector res = 2.0 - x;
NumericVector res = y * 2.0 ;
NumericVector res = 2.0 / y;

// two expressions
NumericVector res = x * y + y / 2.0 ;
NumericVector res = x * ( y - 2.0 ) ;
NumericVector res = x / ( y * y ) ;
```

Binary logical operators

```
// two integer vectors of the same size
NumericVector x ;
NumericVector y ;

// expressions involving two vectors
LogicalVector res = x < y ;
LogicalVector res = x > y ;
LogicalVector res = x <= y ;
LogicalVector res = x >= y ;
LogicalVector res = x == y ;
LogicalVector res = x != y ;

// one vector, one single value
LogicalVector res = x < 2 ;
LogicalVector res = 2 > x;
LogicalVector res = y <= 2 ;
LogicalVector res = 2 != y;

// two expressions
LogicalVector res = ( x + y ) <  ( x*x ) ;
LogicalVector res = ( x + y ) >= ( x*x ) ;
LogicalVector res = ( x + y ) == ( x*x ) ;
```

Unary operators

```
// a numeric vector
NumericVector x ;

// negate x
NumericVector res = -x ;

// use it as part of a numerical expression
NumericVector res = -x * ( x + 2.0 ) ;


// two integer vectors of the same size
NumericVector y ;
NumericVector z ;

// negate the logical expression "y < z"
LogicalVector res = ! ( y < z );
```

Outline

4. Syntactic sugar
   - Motivation
   - Contents
   - Operators
   - Functions
   - Performance

Functions producing a single logical result

```
IntegerVector x = seq_len( 1000 ) ;
all( x*x < 3 ) ;

any( x*x < 3 ) ;


// wrong: will generate a compile error
bool res = any( x < y) ) ;

// ok
bool res = is_true( any( x < y ) )
bool res = is_false( any( x < y ) )
bool res = is_na( any( x < y ) )
```

Functions producing sugar expressions

```
IntegerVector x = IntegerVector::create( 0, 1, NA_INTEGER, 3 ) ;

is_na( x )
all( is_na( x ) )
any( ! is_na( x ) )

seq_along( x )
seq_along( x * x * x * x * x * x * x )

IntegerVector x = seq_len( 10 ) ;

pmin( x, x*x );
pmin( x*x, 2 );

IntegerVector x, y;

ifelse( x < y, x, (x+y)*y );
ifelse( x > y, x, 2 );

sign( xx );
sign( xx * xx );

diff( xx );
```

Mathematical functions

```
IntegerVector x;

abs( x )
exp( x )
log( x )
log10( x )
floor( x )
ceil( x )
sqrt( x )
pow(x, z)     # x to the power of z
```

plus the regular trigonometrics functions and more.

Statistical function d/q/p/r

```
x1 = dnorm(y1, 0, 1);   // density of y1 at m=0, sd=1
x2 = pnorm(y2, 0, 1);   // distribution function of y2
x3 = qnorm(y3, 0, 1);   // quantiles of y3
x4 = rnorm(n, 0, 1);    // 'n' RNG draws of N(0, 1)
```

For beta, binom, caucht, exp, f, gamma, geom, hyper, lnorm, logis, nbeta, nbinom, nbinom_mu, nchisq, nf, norm, nt, pois, t, unif and weibull.

Use something like `RNGScope scope;` to set/reset the RNGs.

Outline

4. Syntactic sugar
   - Motivation
   - Contents
   - Operators
   - Functions
   - Performance

Sugar: benchmarks

expression	sugar	R	R / sugar
`any(x*y<0)`	0.000451	5.17	11450
`ifelse(x<y,x*x,-(y*y))`	1.378	13.15	9.54
`ifelse(x<y,x*x,-(y*y))` (*)	1.254	13.03	10.39
`sapply(x,square)`	0.220	113.38	515.24

Source: `examples/SugarPerformance/` using R 2.13.0, **Rcpp** 0.9.4, `g++-4.5`, Linux 2.6.32, i7 cpu.

$\star$: version includes optimization related to the absence of missing values

Sugar: benchmarks

Benchmarks of the convolution example from Writing R Extensions.

Implementation	Time in millisec	Relative to R API
R API (as benchmark)	234	
Rcpp sugar	158	0.68
`NumericVector::iterator`	236	1.01
`NumericVector::operator[]`	305	1.30
R API *naively*	2199	9.40

Table: Convolution of *x* and *y* (200 values), repeated 5000 times.

Source: `examples/ConvolveBenchmarks/` using R 2.13.0, **Rcpp** 0.9.4, `g++-4.5`, Linux 2.6.32, i7 cpu.

Sugar: Final Example

examples/part2/sugarExample.R

Consider a simple R function of a vector:

```
foo <- function(x) {

    ## sum of
    ##    -- squares of negatives
    ##    -- exponentials of positives
    s <- sum(ifelse( x < 0,  x*x,  exp(x) ))

    return(s)
}
```

Sugar: Final Example

examples/part2/sugarExample.R

Here is one C++ solution:

```
bar <- cxxfunction(signature(xs="numeric"),
                   plugin="Rcpp", body='

  NumericVector x(xs);

  double s = sum( ifelse( x < 0, x*x, exp(x) ));

  return wrap(s);
')
```

Sugar: Final Example

Benchmark from examples/part2/sugarExample.R

```
R> library(compiler)
R> cfoo <- cmpfun(foo)
R> library(rbenchmark)
R> x <- rnorm(1e5)
R> benchmark(foo(x), cfoo(x), bar(x),
+           columns=c("test", "elapsed", "relative",
+                     "user.self", "sys.self"),
+           order="relative", replications=10)
     test elapsed relative user.self sys.self
3  bar(x)   0.033   1.0000      0.03        0
1  foo(x)   0.441  13.3636      0.45        0
2 cfoo(x)   0.463  14.0303      0.46        0
R>
```