Rcpp Masterclass / Workshop
Part II: Rcpp Details

Dirk Eddelbuettel\textsuperscript{1} \quad Romain François\textsuperscript{2}

\textsuperscript{1} Debian Project
\textsuperscript{2} R Enthusiasts

28 April 2011
preceding \textit{R / Finance 2011}
University of Illinois at Chicago
Outline

1 Main Rcpp Classes
   • RObject
     • IntegerVector
     • NumericVector
     • GenericVector
     • DataFrame
     • Function
     • Environments
     • S4
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.
<table>
<thead>
<tr>
<th>Rcpp class</th>
<th>R typeof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer(Vector</td>
<td>Matrix)</td>
</tr>
<tr>
<td>Numeric(Vector</td>
<td>Matrix)</td>
</tr>
<tr>
<td>Logical(Vector</td>
<td>Matrix)</td>
</tr>
<tr>
<td>Character(Vector</td>
<td>Matrix)</td>
</tr>
<tr>
<td>Raw(Vector</td>
<td>Matrix)</td>
</tr>
<tr>
<td>Complex(Vector</td>
<td>Matrix)</td>
</tr>
<tr>
<td>List</td>
<td>list (aka generic vectors) ...</td>
</tr>
<tr>
<td>Expression(Vector</td>
<td>Matrix)</td>
</tr>
<tr>
<td>Environment</td>
<td>environment</td>
</tr>
<tr>
<td>Function</td>
<td>function</td>
</tr>
<tr>
<td>XPtr</td>
<td>externalptr</td>
</tr>
<tr>
<td>Language</td>
<td>language</td>
</tr>
<tr>
<td>S4</td>
<td>S4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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
Let us reimplement (a simpler version of) `prod()` for integer vectors:

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

```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)
```
The loop counter can use the information from the \texttt{IntegerVector} itself:

```r
library(inline)

src <- '
    \texttt{Rcpp::IntegerVector vec(vx);}
    \texttt{int prod = 1;}
    \texttt{for (int i=0; i<vec.size(); i++) {
        prod *= vec[i];
    }\n    return Rcpp::wrap(prod);\n

fun <- \texttt{cxxfunction(signature(vx="integer"),
                        src, plugin="Rcpp")}
fun(1L:10L)
```
We simply access elements by index (but note that the range is over $0 \ldots N - 1$ as is standard for C and C++):

```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)
```
We return the scalar \texttt{int} by using the \texttt{wrap} helper:

library(\texttt{inline})

\begin{verbatim}
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)
\end{verbatim}
As an alternative, the Standard Template Library also allows us a loop-less variant similar in spirit to vectorised R expressions:

```r
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)
```
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:

```r
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
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    1  22  22
2    2  2  44
3    3  3  3

R> fun(1:3L)
   orig  x1  x2
1    1  1  22
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 );
```
NumericMatrix is a specialisation using a dimension attribute:

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

```r
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.
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).
Main Rcpp Classes

- RObject
- IntegerVector
- NumericVector
- GenericVector
- DataFrame
- Function
- Environments
- S4
We can use the `List` type to receive parameters from R. This is an example from the `RcppExamples` package:

```cpp
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:

```cpp
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.
Main Rcpp Classes

- RObject
- IntegerVector
- NumericVector
- GenericVector
- DataFrame
- Function
- Environments
- S4
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.

```r
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()
```
Main Rcpp Classes

- RObject
- IntegerVector
- NumericVector
- GenericVector
- DataFrame
- Function
- Environments
- S4
Functions are another types of SEXP object we can represent:

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

```r
src <- '  Rcpp::Function rt("rt");
  return rt(5, 3);
',

fun <- cxxfunction(signature(),
  src, plugin="Rcpp")
set.seed(42)
fun()
```

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

- RObject
- IntegerVector
- NumericVector
- GenericVector
- DataFrame
- Function

Environments

S4
The Environment class helps us access R environments.

```r
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 environment of the (base) package stats is instatiated, and we access the rnorm() function from it. This is an alternative to accessing build-in functions.
Main Rcpp Classes

- RObject
- IntegerVector
- NumericVector
- GenericVector
- DataFrame
- Function
- Environments
- S4
S4 classes can also be created, or altered, at the C++ level.

```r
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))
```
Extending Rcpp via \texttt{as} and \texttt{wrap}

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

They are declared as

```cpp
// 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);
```
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)
Extending Rcpp via `as` and `wrap`

- Introduction
- Extending wrap
- Extending as
We can declare a new conversion to SEXP operator for class Foo in a header Foo.h before the header Rcpp.h is included.

```cpp
#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.
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 \texttt{wrap}:

```cpp
#include <RcppCommon.h>

// third party library that declares class \texttt{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>
```
We can also declare a partial specialization as the compiler will pick the appropriate overloading:

```cpp
#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>
```
Extending Rcpp via `as` and `wrap`
- Introduction
- Extending wrap
- Extending `as`
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:

```cpp
#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:

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

```cpp
#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.
3 Using Rcpp in your package

- Overview
- Call
- C++ files
- R file
- DESCRIPTION and NAMESPACE
- Makevars and Makevars.win
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
#ifndef _mypackage_RCCPP_HELLO_WORLD_H
#define _mypackage_RCCPP_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
```cpp
#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 ;
}
```
3 Using Rcpp in your package

- Overview
- Call
- C++ files
- R file
  - DESCRIPTION and NAMESPACE
  - Makevars and Makevars.win
The R file makes one call to the one C++ function:

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

```r
useDynLib(mypackage)
extportPattern("^[[:alpha:]]+")
```
Using Rcpp in your package

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

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

```r
## 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()")
)
```

Dirk Eddelbuettel and Romain François

Rcpp Masterclass on 28 April 2011 — Part II: Details
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
Recall the earlier example of a simple (albeit contrived for the purposes of this discussion) R vector expression:

\[
\text{ifelse}(x < y, x \times x, -(y \times y))
\]

which for a given vector \( x \) will execute a simple transformation.

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

```c++
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);
}
```

But with sugar we can simply write it as

```c++
SEXP foo(SEXP xx, SEXP yy) {
    NumericVector x(xx), y(yy);
    return ifelse(x < y, x*x, -(y*y));
}
```
Sugar also gives us things like `sapply` on C++ vectors:

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

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

4 Syntactic sugar

- Motivation
- Contents
- Operators
- Functions
- Performance
Sugar: Overview of Contents

logical operators  \(<, >, \leq, \geq, =, !=\)

arithmetic operators  \(+, -, *, /\)

functions on vectors  \(\text{abs, all, any, ceiling, diag, diff, exp, head, ifelse, is\_na, lapply, pmin, pmax, pow, rep, rep\_each, rep\_len, rev, sapply, seq\_along, seq\_len, sign, sum, tail}\)

functions on matrices  \(\text{outer, col, row, lower\_tri, upper\_tri, diag}\)

statistical functions (dpqr)  \(\text{rnorm, dpois, qlogis, etc ...}\)

More information in the \text{Rcpp-sugar vignette}.\)
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

Dirk Eddelbuettel and Romain François

Rcpp Masterclass on 28 April 2011 — Part II: Details
Functions producing a single logical result

```r
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

```r
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 * 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 );
```

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 trigonometric functions and more.
Statistical function d/q/p/r

\[
\begin{align*}
x_1 &= \text{dnorm}(y_1, 0, 1); \quad \text{// density of } y_1 \text{ at } m=0, \text{ sd}=1 \\
x_2 &= \text{pnorm}(y_2, 0, 1); \quad \text{// distribution function of } y_2 \\
x_3 &= \text{qnorm}(y_3, 0, 1); \quad \text{// quantiles of } y_3 \\
x_4 &= \text{rnorm}(n, 0, 1); \quad \text{// 'n' RNG draws of } N(0, 1)
\end{align*}
\]

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 \texttt{RNGScope scope;} to set/reset the RNGs.
Outline

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

Dirk Eddelbuettel and Romain François
Rcpp Masterclass on 28 April 2011 — Part II: Details
## Sugar: benchmarks

<table>
<thead>
<tr>
<th>expression</th>
<th>sugar</th>
<th>R</th>
<th>R / sugar</th>
</tr>
</thead>
<tbody>
<tr>
<td>any(x*y&lt;0)</td>
<td>0.000451</td>
<td>5.17</td>
<td>11450</td>
</tr>
<tr>
<td>ifelse(x&lt;y, x<em>x, -(y</em>y))</td>
<td>1.378</td>
<td>13.15</td>
<td>9.54</td>
</tr>
<tr>
<td>ifelse(x&lt;y, x<em>x, -(y</em>y)) (*)</td>
<td>1.254</td>
<td>13.03</td>
<td>10.39</td>
</tr>
<tr>
<td>sapply(x, square)</td>
<td>0.220</td>
<td>113.38</td>
<td>515.24</td>
</tr>
</tbody>
</table>

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

★ : version includes optimization related to the absence of missing values
Sugar: benchmarks

Benchmarks of the convolution example from Writing R Extensions.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time in millisec</th>
<th>Relative to R API</th>
</tr>
</thead>
<tbody>
<tr>
<td>R API (as benchmark)</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>Rcpp sugar</td>
<td>158</td>
<td>0.68</td>
</tr>
<tr>
<td><code>NumericVector::iterator</code></td>
<td>236</td>
<td>1.01</td>
</tr>
<tr>
<td><code>NumericVector::operator[]</code></td>
<td>305</td>
<td>1.30</td>
</tr>
<tr>
<td>R API <em>naively</em></td>
<td>2199</td>
<td>9.40</td>
</tr>
</tbody>
</table>

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.
Consider a simple R function of a vector:

```r
foo <- function(x) {
  ## sum of
  ## -- squares of negatives
  ## -- exponentials of positives
  s <- sum(ifelse(x < 0, x*x, exp(x)))

  return(s)
}
```
Here is one C++ solution:

```r
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(foox, 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  foox  0.441  13.3636   0.45      0
    2  cfoo(x)  0.463  14.0303   0.46      0
R>