Higher-Performance R via C++

Part 2: First Steps with Rcpp

Dirk Eddelbuettel
UZH/ETH Zürich R Courses
June 24-25, 2015
Overview
The R API

- R is a C program, and C programs can be extended
- R exposes an API with C functions and MACROS
- R also supports C++ out of the box with .cpp extension
- R provides several calling conventions:
  - `.C()` provides the first interface, is fairly limited, and discouraged
  - `.Call()` provides access to R objects at the C level
  - `.External()` and `.Fortran()` exist but can be ignored
- We will use `.Call()` exclusively
The `.Call` Interface

At the C level, everything is a SEXP, and all `.Call()` access use this interface:

```c
SEXP foo(SEXP x1, SEXP x2){
  ...
}
```

which can be called from R via

```r
.Call("foo", var1, var2)
```
Example: Convolution

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

SEXP convolve2(SEXP a, SEXP b) {
  int na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  a = PROTECT(coerceVector(a, REALSXP));
  b = PROTECT(coerceVector(b, REALSXP));
  na = length(a);
  nb = length(b);
  nab = na + nb - 1;
  ab = PROTECT(allocVector(REALSXP, nab));
  xa = REAL(a);
  xb = REAL(b);
  xab = REAL(ab);
  for (int i = 0; i < nab; i++)
    xab[i] = 0.0;
  for (int i = 0; i < na; i++)
    for (int j = 0; j < nb; j++)
      xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return ab;
}
```
Example: Convolution

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericVector
convolve2cpp(Rcpp::NumericVector a,
             Rcpp::NumericVector b) {
    int na = a.length(), nb = b.length();
    Rcpp::NumericVector ab(na + nb - 1);
    for (int i = 0; i < na; i++)
        for (int j = 0; j < nb; j++)
            ab[i + j] += a[i] * b[j];
    return(ab);
}
```
Rcpp Usage
Rcpp Usage: evalCpp
evalCpp() evaluates a single C++ expression. Includes and dependencies can be declared.

This allows us to quickly check C++ constructs.

```r
library(Rcpp)
evalCpp("2 + 2")  # simple test
```

```r
## [1] 4
```

```r
evalCpp("std::numeric_limits<double>::max()")
```

```r
## [1] 1.797693e+308
```
Rcpp Usage: cppFunction
cppFunction() creates, compiles and links a C++ file, and creates an R function to access it.

```r
cppFunction("int exampleCpp11() {
    auto x = 10;
    return x;
}"
, plugins=c("cpp11"))
exampleCpp11()  # same identifier as C++ function
Rcpp Usage: sourceCpp
sourceCpp() is the actual workhorse behind `evalCpp()` and `cppFunction()`\textsuperscript{.} It is described in more detail in the package vignette Rcpp-attributes.

sourceCpp() builds on and extends `cxxfunction()` from package inline, but provides even more ease-of-use, control and helpers – freeing us from boilerplate scaffolding.

A key feature are the plugins and dependency options: other packages can provide a plugin to supply require compile-time parameters (cf RcppArmadillo, RcppEigen, RcppGSL).
Types
Types: Overview
The RObject can be thought of as a basic class behind many of the key classes in the Rcpp API.

RObject (and our core classes) provide a thin wrapper around SEXP objects

This is sometimes called a *proxy object* as we do not copy the R object.

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

Core classes define several member common functions common to all objects (e.g. isS4(), attributeNames, ...); classes then add their specific member functions.
### Overview of Classes: Comparison

<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><strong>XPtr</strong></td>
<td>externalptr</td>
</tr>
<tr>
<td>Language</td>
<td>language</td>
</tr>
<tr>
<td><strong>S4</strong></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 operations:

- `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
A simpler version of prod() for integer vectors:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
int intVec1a(Rcpp::IntegerVector vec) {
  int prod = 1;
  for (int i=0; i<vec.size(); i++) {
    prod *= vec[i];
  }
  return prod;
}
```
Types: IntegerVector
We can also do this for STL vector types:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
int intVec1b(std::vector<int> vec) {
  int prod = 1;
  for (unsigned int i=0; i<vec.size(); i++) {
    prod *= vec[i];
  }
  return prod;
}
```
Loopless for \texttt{Rcpp::IntegerVector}:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
int intVec2a(Rcpp::IntegerVector vec) {
    int prod = std::accumulate(vec.begin(), vec.end(), 1,
                               std::multiplies<int>())
    return prod;
}
```
Loopless for STL’s `std::vector<int>`:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
int intVec2b(std::vector<int> vec) {
    int prod =
        std::accumulate(vec.begin(),
                        vec.end(), 1,
                        std::multiplies<int>());
    return prod;
}
```
Types: NumericVector
This example generalizes sum of squares by supplying an exponentiation argument:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
double numVecEx1(Rcpp::NumericVector vec,
                 double p = 2.0) {
    double sum = 0.0;
    for (int i=0; i<vec.size(); i++) {
        sum += pow(vec[i], p);
    }
    return sum;
}
```
A second example alters a numeric vector:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::DataFrame numVecEx2(Rcpp::NumericVector xs) {
  Rcpp::NumericVector x1(xs);
  Rcpp::NumericVector x2(Rcpp::clone(xs));
  x1[0] = 22;
  x2[1] = 44;

  return (Rcpp::DataFrame::create(Named("orig", xs),
                                   Named("x1", x1),
                                   Named("x2", x2)));
}
```
Calling the last example with two different arguments:

```r
Rcpp::sourceCpp("code/numVecEx2.cpp")
numVecEx2(c(1.0, 2.0, 3.0))
```

```
## orig x1 x2
## 1 22 22 1
## 2 2 2 44
## 3 3 3 3
```

```r
numVecEx2(c(1L, 2L, 3L))
```

```
## orig x1 x2
## 1 22 22 1
## 2 2 2 44
## 3 3 3 3
```
Constructors

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 given elements
NumericVector y = NumericVector::create(1.0, 2.0);
Types: Matrices
NumericMatrix is a specialisation of NumericVector with a dimension attribute:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericMatrix takeRoot(Rcpp::NumericMatrix mm) {
  Rcpp::NumericMatrix m =
      Rcpp::clone<Rcpp::NumericMatrix>(mm);
  std::transform(m.begin(), m.end(),
                 m.begin(), ::sqrt);
  return m;
}
```
```r
Rcpp::sourceCpp("code/numMatEx1.cpp")

takeRoot(matrix((1:9)*1.0, 3, 3));

```
We prefer Armadillo for math though – more later.

```cpp
#include <RcppArmadillo.h>

// [[Rcpp::export]]
Rcpp::List armafun(arma::mat m1) {
    arma::mat m2 = m1 + m1;
    arma::mat m3 = m1 * 2;
    return Rcpp::List::create(m1, m2);
}
```
Types: Other Vectors
Other Vector Types

- **LogicalVector** very similar to **IntegerVector**: two possible values of a logical, or boolean, type – plus NA.
- **CharacterVector** can be used for vectors of character vectors (“strings”).
- **RawVector** can be used for vectors of raw strings (used eg in serialization).
- **Named** can be used to assign named elements in a vector, similar to R construct `a <- c(foo=3.14, bar=42)`.
- **List** (aka **GenericVector**) is the catch-all, different-types-allowed container, more below.
List types can be used to receive (named) values to R. As lists can be nested, each element type is allowed.

```cpp
double someFunction(Rcpp::List params) {
    std::string method =
        Rcpp::as<std::string>(params["method"]);
    double tolerance =
        Rcpp::as<double>(params["tolerance"]);
    Rcpp::NumericVector startvalues =
        params["startvalues"];

    // ... more code here ...
```
Similarly, List types are convenient for returning multiple values to R.

```cpp
return
Rcpp::List::create(Rcpp::Named("method", method),
    Rcpp::Named("tolerance", tolerance),
    Rcpp::Named("iterations", iterations),
    Rcpp::Named("parameters", parameters));
```
DataFrame can receive and return values.

```cpp
Rcpp::IntegerVector v =
    Rcpp::IntegerVector::create(1, 2, 3);
std::vector<std::string> s =
    { "a", "b", "c" }; // C++11
return Rcpp::DataFrame::create(Rcpp::Named("a") = v,
                                Rcpp::Named("b") = s);
```

But because a `data.frame` is a (internally) a list of vectors, not as easy to subset by rows as in R.
Types: Functions
The Function class can access R functions we pass in:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]

SEXP fun(Rcpp::Function f, SEXP x) {
    return f(x);
}
```
sourceCpp("code/functionEx1.cpp")
fun(sort, sample(1:5, 10, TRUE))

## [1] 1 2 2 2 4 4 5 5 5 5

fun(sort, sample(LETTERS[1:5], 10, TRUE))

## [1] "A" "B" "B" "C" "D" "D" "D" "E" "E" "E"
We can also instantiate functions directly:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericVector fun() {
  Rcpp::Function rt("rt");
  return rt(3, 4);
}
```
```
sourceCpp("code/functionEx2.cpp")
set.seed(42)
fun()

## [1]  2.057339 0.100706 -0.075780

set.seed(42)
rt(3, 4)

## [1]  2.057339 0.100706 -0.075780
```
Types: Environments
#include <Rcpp.h>

// [[Rcpp::export]]

Rcpp::NumericVector fun() {
    Rcpp::Environment stats("package:stats");
    Rcpp::Function rt = stats["rt"];
    return rt(3, Rcpp::Named("df", 4));
}
sourceCpp("code/environmentEx1.cpp")
set.seed(42)
fun()

## [1] 2.057339 0.100706 -0.075780

set.seed(42)
rt(3, 4)

## [1] 2.057339 0.100706 -0.075780
Types: S4
S4 objects can be accessed as well as created.

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::S4 fun(Rcpp::S4 x) {
    x.slot("x") = 42;
    return x;
}
```
sourceCpp("code/s4ex1.cpp")
setClass("S4ex", contains="character",
    representation(x="numeric"))

x <- new("S4ex", "bla", x=10);  x

## An object of class "S4ex"
## [1] "bla"
## Slot "x":
## [1] 10

fun(x)

## An object of class "S4ex"
## [1] "bla"
## Slot "x":
## [1] 42