EXTENDING R WITH C++

MOTIVATION, EXAMPLES, AND CONTEXT

Dirk Eddelbuettel

20 April 2018

Debian / R Project / U of Illinois
OUTLINE
• (Very) Quick R Basics Reminder
• C++ in (way less than) a nutshell
• Extending R with C++ via Rcpp
• A Worked Example
• A Case Study
Why R?
xx <- faithful[, "eruptions"]
fit <- density(xx)
plot(fit)
A Simple Example

density.default(x = xx)

N = 272   Bandwidth = 0.3348

U Illinois Stat 385 Guest Lecture
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)
density.default(x = xx)

N = 272   Bandwidth = 0.3348

Density

N = 272   Bandwidth = 0.3348
So Why R?

R enables us to

- work interactively
- explore and visualize data
- access, retrieve and/or generate data
- summarize and report into pdf, html, ...

making it the key language for statistical computing, and a preferred environment for many data analysts.
R as central point

From any one of

- csv
- txt
- xlsx
- xml, json, ...
- web scraping, ...
- hdf5, netcdf, ...
- sas, stata, spss, ...
- various SQL + NOSQL DBs
- various binary protocols

via

into any one of

- txt
- html
- latex and pdf
- html and js
- word
- shiny
- most graphics formats
- other dashboards
- web frontends
Three Principles (Section 1.1)

Object   Everything that exists in R is an object.
Function  Everything that happens in R is a function call.
Interface Interfaces to other software are part of R.
Three Principles (Section 1.1)

Object  Everything that exists in R is an object.

Function  Everything that happens in R is a function call.

Interface  Interfaces to other software are part of R.
That is new. Or is it?
Algorithm Interface

ABC: general
(PORTRAN)
algorithm

XABC: PORTRAN
subroutine to
provide interface
between ABC
language and/or
utility programs

XABC (INSTR, OUTSTR)

Input INSTR → "X"
"Y"

Pointers/values
Argument Names or
Block

Note: Names are
meaningful to Algorithm,
not necessarily to
language

OUTSTR → "B"

Pointers/values
Types (Nodes)
Result Names

Source: John Chamber, personal communication
This became the system known as “Interface”, a precursor to S and R.
C++
Why C++?

• Asking Google leads to tens of million of hits.
• **Wikipedia**: *C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose, powerful programming language*
• C++ is industrial-strength, vendor-independent, widely-used, and *still evolving*
• In science & research, one of the most frequently-used languages: If there is something you want to use / connect to, it probably has a C/C++ API
• As a widely used language it also has good tool support (debuggers, profilers, code analysis)
Scott Meyers: View C++ as a federation of languages

- C provides a rich inheritance and interoperability as Unix, Windows, ... are all build on C.
- *Object-Oriented C++* (maybe just to provide endless discussions about exactly what OO is or should be)
- *Templated C++* which is mighty powerful; template meta programming unequalled in other languages.
- *The Standard Template Library* (STL) is a specific template library which is powerful but has its own conventions.
- *C++11* and *C++14* (and beyond) add enough to be called a fifth language.

NB: Meyers original list of four languages appeared years before C++11.
Why C++?

• Mature yet current
• Strong performance focus:
  • You don’t pay for what you don’t use
  • Leave no room for another language between the machine level and C++
• Yet also powerfully abstract and high-level
• C++11, C++14, C++17, ... a big deal giving us new language features
• While there are complexities, Rcpp users are mostly shielded
C++ IN TOO LITTLE TIME
Compiled not Interpreted

Need to compile and link

```c
#include <cstdio>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```
Or streams output rather than `printf`

```cpp
#include <iostream>

int main(void) {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```
g++  -o will compile and link

Next: an example with explicit linking of an external library.
```c
#include <cstdio>

#define MATHLIB_STANDALONE
#include <Rmath.h>

int main(void) {
    printf("N(0,1) 95th percentile %9.8f\n",
            qnorm(0.95, 0.0, 1.0, 1, 0));
}
```
We may need to supply:

- header location via -I,
- library location via -L,
- library via -llibraryname

```
g++ -I/usr/include -c qnorm_rmath.cpp
g++ -o qnorm_rmath qnorm_rmath.o -L/usr/lib -lRmath
```
• R is dynamically typed: `x <- 3.14; x <- "foo"` is valid.
• In C++, each variable must be declared before first use.
• Common types are `int` and `long` (possibly with `unsigned`), `float` and `double`, `bool`, as well as `char`.
• No standard string type, though `std::string` is close.
• All these variables types are scalars which is fundamentally different from R where everything is a vector.
• `class` (and `struct`) allow creation of composite types; classes add behaviour to data to form `objects`.
• Variables need to be declared, cannot change
C++ is a Better C

- control structures similar to what R offers: for, while, if, switch
- functions are similar too but note the difference in positional-only matching, also same function name but different arguments allowed in C++
- pointers and memory management: very different, but lots of issues people had with C can be avoided via STL (which is something Rcpp promotes too)
- sometimes still useful to know what a pointer is ...
This is a second key feature of C++, and it is different from S3 and S4.

```cpp
struct Date {
    unsigned int year;
    unsigned int month;
    unsigned int day
};

struct Person {
    char firstname[20];
    char lastname[20];
    struct Date birthday;
    unsigned long id;
};
```
Object-orientation matches data with code operating on it:

```cpp
class Date {
private:
    unsigned int year
    unsigned int month;
    unsigned int date;
public:
    void setDate(int y, int m, int d);
    int getDay();
    int getMonth();
    int getYear();
}
```
Generic Programming and the STL

The STL promotes generic programming.

For example, the sequence container types \texttt{vector}, \texttt{deque}, and \texttt{list} all support

- \texttt{push_back()} to insert at the end;
- \texttt{pop_back()} to remove from the front;
- \texttt{begin()} returning an iterator to the first element;
- \texttt{end()} returning an iterator to just after the last element;
- \texttt{size()} for the number of elements;

but only \texttt{list} has \texttt{push_front()} and \texttt{pop_front()}.

Other useful containers: \texttt{set}, \texttt{multiset}, \texttt{map} and \texttt{multimap}.
Traversal of containers can be achieved via *iterators* which require suitable member functions `begin()` and `end()`:

```cpp
std::vector<double>::const_iterator si;
for (si = s.begin(); si != s.end(); si++)
    std::cout << *si << std::endl;
```
Another key STL part are *algorithms*:

```cpp
double sum = accumulate(s.begin(), s.end(), 0);
```

Some other STL algorithms are

- **find** finds the first element equal to the supplied value
- **count** counts the number of matching elements
- **transform** applies a supplied function to each element
- **for_each** sweeps over all elements, does not alter
- **inner_product** inner product of two vectors
Template programming provides a ‘language within C++’: code gets evaluated during compilation.

One of the simplest template examples is

```cpp
template <typename T>
const T& min(const T& x, const T& y) {
    return y < x ? y : x;
}
```

This can now be used to compute the minimum between two `int` variables, or `double`, or in fact any `admissible type` providing an `operator<()` for less-than comparison.
Another template example is a class squaring its argument:

```cpp
template <typename T>
class square : public std::unary_function<T, T> {
public:
    T operator()(T t) const {
        return t*t;
    }
};
```

which can be used along with STL algorithms:

```cpp
transform(x.begin(), x.end(), square);
```
Books by Meyers are excellent
I also like the (free) C++ Annotations
C++ FAQ
Resources on StackOverflow such as

• general info and its links, eg
• booklist
Some tips:

- Generally painful, old-school `printf()` still pervasive
- Debuggers go along with compilers: `gdb` for `gcc` and `g++`; `lldb` for the clang / llvm family
- Extra tools such as `valgrind` helpful for memory debugging
- “Sanitizer” (ASAN/UBSAN) in newer versions of `g++` and `clang++`
EXTENDING R WITH C++
Rcpp: First steps

Three key functions

- `evalCpp()`
- `sourceCpp()`
- `cppFunction()`
**Basic 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

## [1] 4

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

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

```cpp
cppFunction(""
    int simpleExample() {
        int x = 10;
        return x;
    }
")
simpleExample()  # same identifier as C++ function
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
```
**Basic Usage: sourceCpp()**

`sourceCpp()` is the actual workhorse behind `evalCpp()` and `cppFunction()`. 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`).
Basic Usage: RStudio

![RStudio interface with file new file menu open](image)

- New File
  - New Project...
  - Open File...
  - Recent Files
- Open Project...
- Open Project in New Window...
- Recent Projects
- Save
- Save As...
- Save All
- Print...

Options:
- R Script
- R Markdown...
- Text File
- C++ File
- R Sweave
- R HTML
- R Presentation
- R Documentation
The following file gets created:

```cpp
#include <Rcpp.h>
using namespace Rcpp;

// This is a simple example of exporting a C++ function to R. You can
// source this function into an R session using the Rcpp::sourceCpp
// function (or via the Source button on the editor toolbar). ...

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) { return x * 2; }

// You can include R code blocks in C++ files processed with sourceCpp
// (useful for testing and development). The R code will be automatically
// run after the compilation.

/*** R
 timesTwo(42)
 */
```
So what just happened?

- We defined a simple C++ function
- It operates on a numeric vector argument
- We asked Rcpp to ‘source it’ for us
- Behind the scenes Rcpp creates a wrapper
- Rcpp then compiles, links, and loads the wrapper
- The function is available in R under its C++ name
Package are *the* standard unit of R code organization.

Creating packages with Rcpp is easy; an empty one to work from can be created by `Rcpp.package.skeleton()`

The vignette *Rcpp-packages* has fuller details.

As of April 2018, there are over 1300 packages on CRAN which use Rcpp, and a almost 100 more on BioConductor — with working, tested, and reviewed examples.
Rcpp.package.skeleton() and its derivatives. e.g. RcppArmadillo.package.skeleton() create working packages.

// another simple example: outer product of a vector, returning a matrix

//

// [[Rcpp::export]]
arma::mat rcpparma_outerproduct(const arma::colvec & x) {
    arma::mat m = x * x.t();
    return m;
}

// and the inner product returns a scalar

//

// [[Rcpp::export]]
double rcpparma_innerproduct(const arma::colvec & x) {
    double v = arma::as_scalar(x.t() * x);
    return v;
}
Nice, but does it really work?
Something self-contained

• Let’s talk random numbers!
• We’ll look at a quick generator
• And wrap it in plain C / C++
int getRandomNumber() {
    return 4; // chosen by fair dice roll.
    // guaranteed to be random.
}
/ cf https://xkcd.com/221/
/
"RFC 1149.5 specifies 4 as the "
"standard IEEE-vetted random number."

int getRandomNumber()
{
    return 4; // chosen by fair dice roll
    // guaranteed to be random
}
```cpp
#include <Rcpp.h>
#include <xkcdRng.h>

// [[Rcpp::export]]
int getXkcdRngDraw() {
  return getRandomNumber();
}
```
What Did We Do?

- An unmodified piece of C / C++ code
- A simple interface function
- Rcpp does the rest
A CASE STUDY
A recent blogpost on “finding a needle in a haystack” has a nice story:

```r
options(width=50)
set.seed(1)
haystack <- sample(0:12, size = 2000, replace = TRUE)
needle <- c(2L, 10L, 8L)
haystack[1:60]
```

```
## [1]  3  4  7 11  2 11 12  8  8  0  2  2  8  4 10
## [16]  6  9 12  4 10 12  2  8  1  3  5  0  4 11  4
## [31]  6  7  6  2 10  8 10  1  9  5 10  8 10  7  6
## [46]  10  0  6  9  9  6 11  5  3  0  1  4  6  8  5
```
A recent blogpost on “finding a needle in a haystack” has a nice story:

```r
options(width=50)
set.seed(1)
haystack <- sample(0:12, size = 2000, replace = TRUE)
needle <- c(2L, 10L, 8L)
haystack[1:60]
```

```r
## [1]  3  4  7 11  2 11 12  8  8  0  2  2  8  4 10
## [16]  6  9 12  4 10 12  2  8  1  3  5  0  4 11  4
## [31]  6  7  6  2 10  8 10  1  9  5 10  8 10  7  6
## [46] 10  0  6  9  9  6 11  5  3  0  1  4  6  8  5
```
forloop_find <- function(needle, haystack) {
  n <- length(needle) - 1L
  for (i in seq(haystack)) {
    if (identical(haystack[i:(i+n)], needle)) {
      return(i)
    }
  }
}

forloop_find(needle, haystack)

## [1] 34
lead_find <- function(needle, haystack) {
  v <- haystack == needle[1]
  for (i in seq(2, length(needle))) {
    v <- v +
    (dplyr::lead(haystack, i-1L) == needle[i])
  }
  which(v == length(needle))[1L]
}

lead_find(needle, haystack)

## [1] 34
shift_find <- function(needle, haystack) {
  shifted_haystack <-
  data.table::shift(haystack, type='lead',
                    0:(length(needle)-1))

  v <- Map('==', shifted_haystack, needle)
  v <- Reduce('+', v)
  which(v == length(needle))[1]
}

shift_find(needle, haystack)

## [1] 34
Rcpp::cppFunction('int rcpp_find(NumericVector needle, 
                   NumericVector haystack) {
    int nlen = needle.size(), hlen = haystack.size(), j;
    for (int i = 0; i < (hlen - nlen); i++) {
        for (j = 0; j < nlen; j++) {
            if (needle[j] != haystack[i + j]) break;
        }
        if (j == nlen) return(i+1);
    }
    return(0);
}')
rcpp_find(needle, haystack)
Rcpp::cppFunction(''
 int idiomatiRcpp_find(NumericVector needle,
   NumericVector haystack) {
   NumericVector::iterator it;
   it = std::search(haystack.begin(), haystack.end(),
     needle.begin(), needle.end());
   int pos = it - haystack.begin() + 1;
   if (pos > haystack.size()) pos = -1;
   return(pos);
}'
 idiomaticRcpp_find(needle, haystack)

## [1] 34
```r
R> res <- microbenchmark::microbenchmark(...) # not shown
R> res
Unit: microseconds

<table>
<thead>
<tr>
<th>expr</th>
<th>expr</th>
<th>min</th>
<th>lq</th>
<th>mean</th>
<th>median</th>
<th>uq</th>
<th>max</th>
<th>neval</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollapply_find(needle, haystack)</td>
<td>5829.484</td>
<td>6355.7290</td>
<td>6918.75051</td>
<td>6719.5825</td>
<td>7114.4770</td>
<td>37348.915</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>vapply_find(needle, haystack)</td>
<td>1230.373</td>
<td>1338.3275</td>
<td>1519.16471</td>
<td>1419.7170</td>
<td>1491.9485</td>
<td>31687.188</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>grep_find(needle, haystack)</td>
<td>535.059</td>
<td>556.3545</td>
<td>592.86697</td>
<td>572.5000</td>
<td>597.6460</td>
<td>1396.680</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>forloop_find(needle, haystack)</td>
<td>169.751</td>
<td>189.5370</td>
<td>213.39386</td>
<td>200.6070</td>
<td>210.2785</td>
<td>1151.483</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>lead_find(needle, haystack)</td>
<td>47.571</td>
<td>55.4575</td>
<td>61.54025</td>
<td>59.2370</td>
<td>62.3445</td>
<td>331.499</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>shift_find(needle, haystack)</td>
<td>37.939</td>
<td>47.5780</td>
<td>55.14043</td>
<td>52.3475</td>
<td>58.5400</td>
<td>268.533</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>pile_find(needle, haystack)</td>
<td>13.883</td>
<td>15.7375</td>
<td>17.10174</td>
<td>16.5590</td>
<td>17.4175</td>
<td>45.757</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>sieved_find(needle, haystack)</td>
<td>7.587</td>
<td>9.4575</td>
<td>10.82973</td>
<td>10.4355</td>
<td>11.3620</td>
<td>29.978</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>boyer_moore(needle, haystack)</td>
<td>3.197</td>
<td>4.7035</td>
<td>6.06770</td>
<td>5.6540</td>
<td>6.5950</td>
<td>89.414</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>rcpp_find(needle, haystack)</td>
<td>2.579</td>
<td>3.6805</td>
<td>4.86756</td>
<td>4.5465</td>
<td>5.3570</td>
<td>27.765</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>idiomaticrcpp_find(needle, haystack)</td>
<td>2.183</td>
<td>3.5230</td>
<td>4.62004</td>
<td>4.3555</td>
<td>5.1945</td>
<td>24.235</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
```
ggplot2::autoplot(res)
CONCLUSION
Takeaways on *Extending R with C++*

- clearly possible as the tooling helps greatly
- natural as interfaces are a normal part of R
- not too hard, though balancing two languages
- rewarding in terms of performance
- always measure and profile
Thank you!

slides  http://dirk.eddelbuettel.com/presentations/
web    http://dirk.eddelbuettel.com/
mail   dirk@eddelbuettel.com
github @eddelbuettel
twitter @eddelbuettel