Motivation

Why R? : Programming with Data from 1997 to 2016

Thanks to John Chambers for sending me high-resolution scans of the covers of his books.
A Simple Example

xx <- faithful[, "eruptions"]
fit <- density(xx)
plot(fit)

A Simple Example - Refined

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)
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 has always been extensible via

- C via a bare-bones interface described in *Writing R Extensions*
- Fortran which is also used internally by R
- Java via *rJava* by Simon Urbanek
- C++ but essentially at the bare-bones level of C

So while *in theory* this always worked – it was tedious *in practice*

**Why Extend R?**

Chambers (2008), opens Chapter 11 *Interfaces I: Using C and Fortran*:

*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.*
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 focus for us in this workshop.
- **new things:** We can bind to libraries and tools that would otherwise be unavailable in R
- **references:** Chambers quote from 2008 foreshadowed the work on *Reference Classes* now in R and built upon via Rcpp Modules, Rcpp Classes (and also RcppR6)

**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 is a big deal giving us new language features
- While there are complexities, Rcpp users are mostly shielded

---

**Interface Vision**

R offers us the best of both worlds:

- **Compiled** code with
  - Access to proven libraries and algorithms in C/C++/Fortran
  - Extremely high performance (in both serial and parallel modes)
- **Interpreted** code with
  - An accessible high-level language made for *Programming with Data*
  - An interactive workflow for data analysis
  - Support for rapid prototyping, research, and experimentation

---

**Why Rcpp?**

- **Easy to learn** as it really does not have to be that complicated
- **Easy to use** as it avoids build and OS system complexities thanks to the R infrastructure
- **Expressive** as it allows for *vectorised C++* using *Rcpp Sugar*
- **Speed gains** for a variety of tasks Rcpp excels precisely where R struggles: loops, function calls, ...
- **Extensions** greatly facilitates access to external libraries using eg *Rcpp modules*
Consider a function defined as
\[
f(n) = \begin{cases} 
  n & \text{when } n < 2 \\
  f(n-1) + f(n-2) & \text{when } n \geq 2 
\end{cases}
\]

R implementation and use:

```r
f <- function(n) {
  if (n < 2) return(n)
  return(f(n-1) + f(n-2))
}
```

## Using it on first 11 arguments
```
sapply(0:10, f)
## [1] 0 1 1 2 3 5 8 13 21 34 55
```

### Timing:

```r
library(rbenchmark)
benchmark(f(10), f(15), f(20))[,1:4]
```

<table>
<thead>
<tr>
<th>test</th>
<th>replications</th>
<th>elapsed</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(10)</td>
<td>100</td>
<td>0.020</td>
<td>1.0</td>
</tr>
<tr>
<td>f(15)</td>
<td>100</td>
<td>0.222</td>
<td>11.1</td>
</tr>
<tr>
<td>f(20)</td>
<td>100</td>
<td>2.490</td>
<td>124.5</td>
</tr>
</tbody>
</table>

A C or C++ solution can be equally simple

```c
int g(int n) {
  if (n < 2) return(n);
  return(g(n-1) + g(n-2));
}
```

But how do we call it from R?
**Speed Example in C / C++**

But Rcpp makes this *much* easier:

```r
Rcpp::cppFunction("int g(int n) {
  if (n < 2) return(n);
  return(g(n-1) + g(n-2)); }")
sapply(0:10, g)
```

```r
## [1] 0 1 1 2 3 5 8 13 21 34 55
```

**Another Angle on Speed**

Run-time performance is just one example.

*Time to code* is another metric.

We feel quite strongly that helps you code more succinctly, leading to fewer bugs and faster development.

A good environment helps. RStudio integrates R and C++ development quite nicely (eg the compiler error message parsing is very helpful) and also helps with package building.

**Speed Example Comparing R and C++**

```r
timing:
Rcpp::cppFunction("int g(int n) {
  if (n < 2) return(n);
  return(g(n-1) + g(n-2)); }")
library(rbenchmark)
benchmark(f(25), g(25), order="relative")[1:4]
```

```r
## test replications elapsed relative
## 2 g(25) 100 0.066 1.000
## 1 f(25) 100 28.379 429.985
```

A nice gain of a few orders of magnitude.

**(Very Brief) C++ Basics**
Hands-on C++

- C++ Basics
- Debugging
- Best Practices

and then on to Rcpp itself

C++ Basics

Compiled not Interpreted

Need to compile and link

```cpp
#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

We will now look at an examples with explicit linking.
```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`

```bash
g++ -I/usr/include -c qnorm_rmath.cpp
g++ -o qnorm_rmath qnorm_rmath.o -L/usr/lib -lRmath
```

### Statically Typed

- 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

### A Better C

**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 does it differently from S3 and S4.

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

```cpp
struct Person {
    char firstname[20];
    char lastname[20];
    struct Date birthday;
    unsigned long id;
};
```

Object-orientation in the C++ sense 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();
};
```

The STL promotes generic programming. For example, the sequence container types `vector`, `deque`, and `list` all support

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

but only `list` has `push_front()` and `pop_front()`.

Other useful containers: `set`, `multiset`, `map` and `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);
```

Further Reading

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++`

**Getting Started with Rcpp**

**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
```

Some Tips

- Version control: highly recommended to become familiar with `git` or `svn`
- Editor: *in the long-run*, recommended to learn productivity tricks for one editor: emacs, vi, eclipse, RStudio, …
**cppFunction()** creates, compiles and links a C++ file, and creates an R function to access it.

```cpp
cppFunction("    int exampleCpp11() {
        auto x = 10;
        return x;
    }", plugins=c("cpp11"))
```

```r
evalRcpp(exampleCpp11())  # same identifier as C++ function
```

**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).

**RStudio**

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).

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)
*/
```
Basic Usage: RStudio (Cont’d)

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

Basic Usage: Packages

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 July 2016, there are almost 700 packages on CRAN which use Rcpp, and a further 72 on BioConductor — with working, tested, and reviewed examples.

Packages and Rcpp

Best way to organize R code with Rcpp is via a package:

Rcpp.package.skeleton() and its derivatives. e.g. `RcppArmadillo.package.skeleton()` create working packages.

```cpp
// 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;
}
```
**Two ways to link to external libraries**

- *With linking of libraries:* Do what RcppGSL does and use hooks in the package startup to store compiler and linker flags, pass to environment variables.

- *With C++ template headers only:* Do what RcppArmadillo and other do and just point to the headers.

More details in extra vignettes.

---

**The R API**

*In a nutshell:*

- 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 every `.Call()` access uses this interface pattern:

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

which can be called from R via

```r
.Call("foo", var1, var2)
```

Note that we need to compile, and link, and load, this manually in ways which are OS-dependent.
# RObject

- 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 type of</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 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

IntegerVector: A first example

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

```c++
#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;
}
```

We can also do this for STL vector types:

```c++
#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;
}
```
**INTEGER VECTOR: LOOPLESS**

For **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;
}
```

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;
}
```

**NUMERIC VECTOR: A FIRST EXAMPLE**

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;
}
```

**NUMERIC VECTOR: A SECOND EXAMPLE**

A second example alters a numeric vector:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
NumericVector f(NumericVector m) {
  m(0) = 0;
  return m;
}
```
Calling the last example with an integer vector:

Rcpp::sourceCpp("code/numVecEx3.cpp")

\[
x \leftarrow 1:3 \\
\text{# same as c(1L, 2L, 3L)}
\]

\[
\text{print(data.frame(x=x, fx=f(x)), row.names=FALSE)}
\]

```
## x fx
## 1 0
## 2 2
## 3 3
```

We pass `x` as a SEXP which is a pointer.
Use `Rcpp::clone()` for deep copy.

**Constructors**

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

// cloning (deep copy)
NumericVector `z = clone(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);`

**NumericMatrix**

`NumericMatrix` is a specialisation of `NumericVector` with a dimension attribute:

```
#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;
}
```

**Calling the last example with a numeric vector:**

\[
x \leftarrow \text{c}(1.0, 2.0, 3.0)
\]

\[
\text{print(data.frame(x=x, fx=f(x)), row.names=FALSE)}
\]

```
## x fx
## 0 0
## 2 2
## 3 3
```

We pass `x` as a SEXP which is a pointer.
Use `Rcpp::clone()` for deep copy.
Rcpp::sourceCpp("code/numMatEx1.cpp")

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

## [,1] [,2] [,3]
## [1,]  1.000000  2.000000  2.645751
## [2,]  1.414214  2.236068  2.828427
## [3,]  1.732051  2.449490  3.000000

We prefer Armadillo for math though – more later.

// [[Rcpp::depends(RcppArmadillo)]]

#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);
}

Other Vector Types

Quick List:

- **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.

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 ...
}
**GenericVector**

Similarly, *List* types can return 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**

*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.

**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);
}
```

```r
sourceCpp("code/functionEx1.cpp")
fun(sort, sample(1:5, 10, TRUE))
## [1] 1 1 2 3 4 5 5 5 5 5
fun(sort, sample(LETTERS[1:5], 10, TRUE))
## [1] "A" "A" "B" "C" "C" "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);
}
```

```r
set.seed(42)
fun()
## [1] 2.057339 0.100706 -0.075780
```

```cpp
// [[Rcpp::export]]
Rcpp::NumericVector fun() {
  Rcpp::Environment stats("package:stats");
  Rcpp::Function rt = stats["rt"];
  return rt(3, Rcpp::Named("df", 4));
}
```

```r
set.seed(42)
rt(3, 4)
## [1] 2.057339 0.100706 -0.075780
```
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;
}
```

```
library(methods); sourceCpp("code/s4ex1.cpp")
setClass("S4ex", contains="character",
    representation(x="numeric"))
x <- new("S4ex", "bla", x=10);  x
```

```r
fun(x)
```

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

As of July 2016, almost 700 packages on CRAN use Rcpp

Single biggest “application” is RcppArmadillo for linear algebra

---

**Applications**

As of July 2016, almost 700 packages on CRAN use Rcpp

Single biggest “application” is RcppArmadillo for linear algebra
**What is Armadillo?**

- Armadillo is a C++ linear algebra library (matrix maths) aiming towards a good balance between speed and ease of use.
- The syntax is deliberately similar to Matlab.
- Integer, floating point and complex numbers are supported.
- A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporaries.
- Useful for conversion of research code into production environments, or if C++ has been decided as the language of choice, due to speed and/or integration capabilities.

**Armadillo Highlights**

- Provides integer, floating point and complex vectors, matrices and fields (3d) with all the common operations.
- Very good documentation and examples
  - website,
  - technical report (Sanderson, 2010)
  - CSDA paper (Sanderson and Eddelbuettel, 2014)
  - JOSS paper (Sanderson and Curtin, 2016).
- Modern code, building on / extending earlier matrix libraries.
- Responsive and active maintainer, frequent updates.
- Used eg by MLPACK, see Curtin et al (JMLR, 2013).

**RcppArmadillo Highlights**

- Template-only builds—no linking, and available wherever R and a compiler work (but Rcpp is needed)
- Easy to use, just add `LinkingTo: RcppArmadillo, Rcpp` to `DESCRIPTION` (i.e. no added cost beyond Rcpp)
- Really easy from R via Rcpp and automatic converters
- Frequently updated, widely used


```cpp
#include <RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]
arma::vec getEigenValues(arma::mat M) {
    return arma::eig_sym(M);
}
```

---

**FastLm Case Study**

Faster Linear Model with FastLm

- Implementations of `fastLm()` have been a staple during development of Rcpp.
- First version was in response to a question by Ivo Welch on `r-help`.
- Request was for a fast function to estimate parameters – and their standard errors – from a linear model.
- It used GSL functions to estimate $\hat{\beta}$ as well as its standard errors $\hat{\sigma}$ – as `lm.fit()` in R only returns the former.
- It has since been reimplemented for RcppArmadillo and RcppEigen.

```r
Rcpp::sourceCpp("code/arma_eigenvalues.cpp")
M <- cbind(c(1,-1), c(-1,1))
getEigenValues(M)
```

```
## [,1]
## [1,] 0
## [2,] 2
```
### Initial FastLm

```c++
#include <RcppArmadillo.h>

extern "C" SEXP fastlm(SEXP Xs, SEXP ys) {
try {
Rcpp::NumericVector yr(ys);  // creates Rcpp vector from SEXP
Rcpp::NumericMatrix Xr(Xs);  // creates Rcpp matrix from SEXP
int n = Xr.nrow(), k = Xr.ncol();
a arma::mat X(Xr.begin(), n, k, false);  // reuses memory, avoids extra copy
arma::colvec y(yr.begin(), yr.size(), false);
arma::colvec coef = arma::solve(X, y);  // fit model y ~ X
arma::colvec resid = y - X*coef;  // residuals
double sig2 = arma::inner_product(resid.begin(), resid.end(), resid.begin(), arma::vec(0.0)/(n - k));
a arma::colvec std_err = arma::sqrt(sig2*arma::diagvec(arma::inv(trans(X)*X)));
return Rcpp::List::create(Rcpp::Named("coefficients") = coef,
                          Rcpp::Named("stderr") = std_err,
                          Rcpp::Named("df.residual") = n - k);
} catch (std::exception &ex) {
foward_exception_to_r( ex );
}
return R_NilValue;  // -Wall
}
```

### Newer Version

```c++
#include <RcppArmadillo.h>

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
List fastLm(NumericVector yr, NumericMatrix Xr) {
int n = Xr.nrow(), k = Xr.ncol();
colvec y(yr.begin(), yr.size(), false);
colvec resid = y - X*coef;
double sig2 = arma::as_scalar(trans(resid)*resid/(n-k));
colvec stderrest = arma::sqrt(sig2 * arma::diagvec( arma::inv(trans(X)*X) ));
return List::create(Named("coefficients") = coef,
                    Named("stderr") = stderrest,
                    Named("df.residual") = n - k);
}
```

### Interface Changes

Convenient, yet incurs an additional copy. Next variant uses two steps, but only a pointer to objects is copied:

```c++
arma::colvec y = Rcpp::as<arma::colvec>(ys);
arma::mat X = Rcpp::as<arma::mat>(Xs);
```

Better if performance is a concern. But now RcppArmadillo has efficient const references too.
## continued from above

### VAR(1) in R

Simulating a VAR(1) system of $k$ variables:

$$X_t = X_{t-1}B + E_t$$

where $X_t$ is a row vector of length $k$, $B$ is a $k$ by $k$ matrix and $E_t$ is a row of the error matrix of $k$ columns.

We use $k = 2$ for this example.

## VAR(1) in 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, errors) {
  simdata <- matrix(0, nrow=errors), ncol=errors)
  for (row in 2:nrow(errors)) {
    simdata[row,] = coeff %*% simdata[(row-1),] + errors[row,
  }
  return(simdata)
}
rData <- rSim(a, e) # generated by R
```
VAR(1) in C++

```cpp
arma::mat rcppSim(const arma::mat& coeff,
                  const arma::mat& errors) {
    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) * coeff +
                           errors.row(row);
    }
    return simdata;
}
```

Benchmark

```r
library(rbenchmark)
Rcpp::sourceCpp("code/arma_var1.cpp")
```

## R> a <- matrix(c(0.5, 0.1, 0.1, 0.5), 2, 2)
## R> e <- matrix(rnorm(10000), ncol = 2)
## R> head(rcppSim(a, e))
## [,1]      [,2]
## [1,] 0.0000000 0.0000000
## [2,] 0.8915759 1.1792437
## [3,] 0.3858789 1.0815238
## [4,] 2.0054965 1.0828000
## [5,] -0.1754779 0.9469223
## [6,] -0.4886083 1.0468917
```

Kalman Filter Case Study

The position of an object is estimated based on past values of $6 \times 1$ state vectors $X$ and $Y$ for position, $V_x$ and $V_y$ for speed, and $A_x$ and $A_y$ for acceleration.

Position updates as a function of the speed

$$X = X_0 + V_x dt$$ and $$Y = Y_0 + V_y dt,$$

which is updated as a function of the (unobserved) acceleration:

$$V_x = V_{x,0} + A_x dt$$ and $$V_y = V_{y,0} + A_y dt.$$
Matlab Code: \texttt{kalmanM.m} with Loop

```matlab
function Y = kalmanM(pos)
dt=1;
Q = eye(6);
R = 1000 * eye(2); % measurement noise
H = zeros(2,6);
% Initialize measurement matrix
y = zeros(numPts,2);
P_est = zeros(6,6);
% Initial state and covariance
for idx = 1:numPts
    x = pos(idx,:);
    x_est = A * x; % x_est = [x,y,Vx,Vy,Ax,Ay]'
    p_est = A * P_est * A' + Q;
    S = H * P_est * H' + R;
    B = H * P_est;
    kalmanGain = (H * B).'
    % kalmanGain = (H * B).'
    x_est = x_est + kalmanGain * (y - H * x_est); % estimate state and covariance
    p_est = P_est - kalmanGain * H * P_est;
    y(idx, :) = H * x_est;
end
end % of the function
```

Now in R

```r
kalmanR <- function(pos) {
  kalmanfilter <- function(z) {
    xprd <- A %*% xest # predicted state and covariance
    pprd <- A %*% p_est %*% t(A) + Q
    S <- H %*% t(pprd) %*% t(H) + R
    B <- H %*% t(pprd)
    kalmangain <- t(solve(H,H)) %*% (I - H %*% xprd) # kalmanGain
    p_est <- pprd - kalmangain %*% H %*% pprd
    xest <- xprd + kalmangain %*% (y - H %*% xprd)
    y <- H %*% xest
    invisible(y)
  }
  dt <- 1
  A <- matrix(c( 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt), nrow = 6, byrow=TRUE)
  R <- diag(6)
  H <- matrix(c(1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1), nrow = 5, byrow=TRUE)
  y <- matrix(c(pos), ncol = 2, nrow = 5, byrow=TRUE)
  xest <- matrix(0, 6, 1)
  for (i in 1:N) y[i,] <- kalmanfilter(t(pos[i,],drop=FALSE))
  invisible(y)
}
```

Improved in R

```r
KalmanR <- function(pos) {
  kalmanfilter <- function(z) {
    xprd <- A %*% xest # predicted state and covariance
    pprd <- A %*% p_est %*% t(A) + Q
    S <- H %*% t(pprd) %*% t(H) + R # estimation
    B <- H %*% t(pprd)
    kalmanGain <- t(solve(H,H)) %*% (I - H %*% xprd) # estimate state and covariance
    p_est <- pprd - kalmanGain %*% H %*% pprd
    xest <- xprd + kalmanGain %*% (y - H %*% xprd)
    y <- H %*% xest # compute the estimated measurements
    invisible(y)
  }
  dt <- 1
  A <- matrix(c( 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt, 0, 0, 0, 1, 0, dt), nrow = 6, byrow=TRUE)
  R <- diag(6)
  H <- matrix(c(1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1), nrow = 5, byrow=TRUE)
  y <- matrix(c(pos), ncol = 2, nrow = 5, byrow=TRUE)
  xest <- matrix(0, 6, 1)
  for (i in 1:N) y[i,] <- kalmanfilter(t(pos[i,],drop=FALSE))
 invisible(y)
}
```

Now in C++

```cpp
#include <Rcpp.h>

using namespace Rcpp;

class Kalman {
private:
  mat A, H, Q, R, xest, pest;
  double dt;
public:
  // Constructor, sets up data structures
  Kalman(): dt(1.0) {
    A.eye(6,6);
    A(0,2) = A(2,4); A(3,5) = dt;
    H.zeros(2,6);
    H(0,0) = H(1,1) = 1.0;
    Q.eye(6,6);
    R = 1000 * eye(2,2);
    xest.zeros(6,1);
    pest.zeros(6,6);
  }
  // cont. below
```
And now in C++

// sole member function: estimate model

mat estimate(const mat & Z) {
  unsigned int n = Z.n_rows, k = Z.n_cols;
  mat Y = zeros(n, k);
  mat xprd, pprd, S, B, kalmangain;
  colvec z, y;

  for (unsigned int i = 0; i < n; i++) {
    z = Z.row(i).t();
    // predicted state and covariance
    xprd = A * xest;
    pprd = A * pest * A.t() + Q;
    // estimation
    S = H * pprd.t() * H.t() + R;
    B = H * pprd.t();
    kalmangain = (solve(S, B)).t();
    // estimated state and covariance
    xest = xprd + kalmangain * (z - H * xprd);
    pest = pprd - kalmangain * H * pprd;
    // compute the estimated measurements
    y = H * xest;
    Y.row(i) = y.t();
  }

  return Y;
}

And the call:

// [[Rcpp::export]]
mat KalmanCpp(mat Z) {
  Kalman K;
  mat Y = K.estimate(Z);
  return Y;
}

Benchmark

library(rbenchmark)
Rcpp::sourceCpp("code/kalman.cpp")
source("code/kalman.R")
p <- as.matrix(read.table("code/pos.txt",
                        header=FALSE,
                        col.names=c("x","y")))
benchmark(KalmanR(p), FirstKalmanR(p), KalmanCpp(p),
          order="relative", replications=500)[,1:4]

## test replications elapsed relative
## 3 KalmanCpp(p) 500  11.445  1.000
## 1 KalmanR(p) 500  23.513  2.054
## 2 FirstKalmanR(p) 500  30.627  2.676

Reproduced Figure

![Reproduced Figure](image-url)
RcppGSL is a convenience wrapper for accessing the GNU GSL, particularly for vector and matrix functions.

- Given that the GSL is a C library, we need to
  - do memory management and free objects (or let C++ do it for us as in recent versions of RcppGSL)
  - arrange for the GSL libraries to be found
- RcppGSL may still be a convenient tool for programmers more familiar with C than C++ wanting to deploy GSL algorithms.

GSL Vector Norm Example (old)

```c
#include <RcppGSL.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

// [[Rcpp::depends(RcppGSL)]]

// [[Rcpp::export]]
Rcpp::NumericVector colNorm_old(Rcpp::NumericMatrix NM) {
  // this conversion involves an allocation
  RcppGSL::matrix<double> M = Rcpp::as<RcppGSL::matrix<double>>(NM);
  int k = M.ncol();
  Rcpp::NumericVector n(k); // to store results
  for (int j = 0; j < k; j++) {
    RcppGSL::vector_view<double> colview = gsl_matrix_column(M, j);
    n[j] = gsl_blas_dnrm2(colview);
  }
  M.free();
  return n; // return vector
}
```

GSL Vector Norm Example (New)

```c
#include <RcppGSL.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

// [[Rcpp::depends(RcppGSL)]]

// newest version using typedefs and const &

// [[Rcpp::export]]
Rcpp::NumericVector colNorm(const RcppGSL::Matrix & G) {
  int k = G.ncol();
  Rcpp::NumericVector n(k); // to store results
  for (int j = 0; j < k; j++) {
    RcppGSL::VectorView colview = gsl_matrix_const_column(G, j);
    n[j] = gsl_blas_dnrm2(colview);
  }
  return n; // return vector
}
```

GSL Vector Norm Example

```r
Rcpp::sourceCpp("code/gslNorm.cpp")
set.seed(42)
M <- matrix(rnorm(25), 5, 5)
colNorm(M) # via GSL
## [1] 1.701241 2.526438 2.992635 3.903917 2.892030
```

```r
apply(M, 2, function(x) sqrt(sum(x^2))) # via R
## [1] 1.701241 2.526438 2.992635 3.903917 2.892030
```
The example comes from Section 39.7 of the GSL Reference manual, and constructs a data set from the curve
\[ y(x) = \cos(x) \exp(-x/10) \] on the interval \([0, 15]\) with added Gaussian noise — which is then fit via linear least squares using a cubic B-spline basis functions with uniform breakpoints.

Obviously all this could be done in R too as R can both generate data, and fit models including (B-)splines. But the point to be made here is that we can very easily translate a given GSL program (thanks to RcppGSL), and get it into R with ease thanks to Rcpp and Rcpp attributes.

```cpp
// [[Rcpp::depends(RcppGSL)]]
#include <RcppGSL.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_statistics.h>

const int N = 200; // number of data points to fit
const int NCOEFFS = 12; // number of fit coefficients
const int NBREAK = (NCOEFFS - 2); // ncoeffs + 2 - k = ncoeffs - 2 as k = 4

// [[Rcpp::export]]
Rcpp::List genData() {
  const size_t n = N;
  size_t i;
  RcppGSL::Vector w(n), x(n), y(n);
  gsl_rng_env_setup();
  gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
  // ...
  return Rcpp::DataFrame::create(Rcpp::Named("x") = x,
                                  Rcpp::Named("y") = y,
                                  Rcpp::Named("w") = w);
}
```

```cpp
for (i = 0; i < n; ++i) {
  // this is the data to be fitted
  double xi = (15.0 / (N - 1)) * i;
  double yi = cos(xi) * exp(-0.1 * xi);
  double sigma = 0.1 * yi;
  double dy = gsl_ran_gaussian(r, sigma);
  double y = yi + dy;
  x[i] = xi;
  y[i] = y;
  w[i] = 1.0 / (sigma * sigma);
}
gsl_rng_free(r);
```

```cpp
// [[Rcpp::export]]
Rcpp::List fitData(Rcpp::DataFrame D) {
  const size_t ncoeffs = NCOEFFS;
  const size_t nbreak = NBREAK;
  const size_t n = N;
  size_t i, j;
  RcppGSL::Vector y = D["y"]; // access columns by name,
  RcppGSL::Vector x = D["x"]; // assigning to GSL vectors
  RcppGSL::Vector w = D["w"];
  gsl_bspline_workspace *bw;
  RcppGSL::Vector c(ncoeffs);
  RcppGSL::Vector B(ncoeffs);
  RcppGSL::Matrix X(n, ncoeffs);
  RcppGSL::Matrix cov(ncoeffs, ncoeffs);
  gsl_multifit_linear_workspace *mw;
  double chisq, Rsq, dof, tss;
  bw = gsl_bspline_alloc(4, nbreak); // allocate a cubic bspline workspace (k = 4)
  mw = gsl_multifit_linear_alloc(n, ncoeffs);
  gsl_bspline_knots_uniform(0.0, 15.0, bw); // use uniform breakpoints on \[0, 15\]
```
for (i = 0; i < n; ++i) { // construct the fit matrix X
double xi = x[i];
gsl_bspline_eval(xi, B, bw); // compute B_j(xi) for all j
for (j = 0; j < ncoeffs; ++j) { // fill in row i of X
double Bj = B[j];
X(i,j) = Bj;
}
}
gsl_multifit_wlinear(X, w, y, c, cov, &chisq, mw); // do the fit
dof = n - ncoeffs;
tss = gsl_stats_wtss(w->data, 1, y->data, 1, y->size);
Rsq = 1.0 - chisq / tss;

Rcpp::sourceCpp("bSpline.cpp")

dat <- genData() # generate the data
fit <- fitData(dat) # fit the model
X <- fit[['X']] # extract vectors
Y <- fit[['Y']]  
par(mar=c(3,3,1,1))
plot(dat[,"x"], dat[,"y"], pch=19, col="#00000044")
lines(X, Y, col="orange", lwd=2)
Overview

- The Rcpp Gallery at http://gallery.rcpp.org provides over one hundred ready-to-run and documented examples.
- It is built on a blog-alike backend in a repository hosted at GitHub.
- You can clone the repository, or just download examples one-by-one.

**Cumulative Sum:** vector-cumulative-sum/

A basic looped version:

```cpp
#include <Rcpp.h>
#include <numeric>
using namespace Rcpp;
// [[Rcpp::export]]

NumericVector cumsum1(NumericVector x) {
  double acc = 0; // init an accumulator var
  NumericVector res(x.size()); // init result vector
  for (int i = 0; i < x.size(); i++){
    acc += x[i];
    res[i] = acc;
  }
  return res;
}
```

An STL variant:

```cpp
#include <Rcpp.h>
#include <numeric>
using namespace Rcpp;
// [[Rcpp::export]]

NumericVector cumsum2(NumericVector x) {
  // initialize the result vector
  NumericVector res(x.size());
  std::partial_sum(x.begin(), x.end(),
                   res.begin());
  return res;
}
```
Cumulative Sum: `vector-cumulative-sum/`

```cpp
Sugar:

// [[Rcpp::export]]
NumericVector cumsum3(NumericVector x) {
    return cumsum(x); // compute + return result
}
```

Calling an R Function: `r-function-from-c++/`

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

// [[Rcpp::export]]
NumericVector callFunction(NumericVector x, Function f) {
    NumericVector res = f(x);
    return res;
}
```

Vector Subsetting: `subsetting/`

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

// [[Rcpp::export]]
NumericVector positives(NumericVector x) {
    return x[x > 0];
}

// [[Rcpp::export]]
List first_three(List x) {
    IntegerVector idx = IntegerVector::create(0, 1, 2);
    return x[idx];
}

// [[Rcpp::export]]
List with_names(List x, CharacterVector y) {
    return x[y];
}
```

Vector Subsetting: `armadillo-subsetting/`

```cpp
#include <RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]
arma::mat matrixSubset(arma::mat M) {
    arma::umat a = trans(M) > M;
    arma::mat N = arma::conv_to<arma::mat>::from(a);
    return N;
}

// [[Rcpp::export]]
arma::vec matrixSubset2(arma::mat M) {
    arma::mat Z = M * M.t();
    arma::vec v = Z.elem( arma::find( Z >= 100 ) );
    return v;
}
```
**Boost via BH: a-first-boost-example/**

```cpp
// [[Rcpp::depends(BH)]]
#include <Rcpp.h>
#include <boost/math/common_factor.hpp>

// [[Rcpp::export]]
int computeGCD(int a, int b) {
    return boost::math::gcd(a, b);
}

// [[Rcpp::export]]
int computeLCM(int a, int b) {
    return boost::math::lcm(a, b);
}
```

**Boost via BH: using-boost-with-bh/**

```cpp
// [[Rcpp::depends(BH)]]
#include <Rcpp.h>
// One include file from Boost
#include <boost/date_time/gregorian/gregorian_types.hpp>

using namespace boost::gregorian;

// [[Rcpp::export]]
Rcpp::Date getIMMDate(int mon, int year) {
    // compute third Wednesday of given month / year
    date d = nth_day_of_the_week_in_month(nth_day_of_the_week_in_month::third,
                                         Wednesday, mon).get_date(year);
    date::ymd_type ymd = d.year_month_day();
    return Rcpp::Date(ymd.year, ymd.month, ymd.day);
}
```

**Boost via BH: boost-regular-expressions/**

```cpp
NB: Use Sys.setenv("PKG_LIBS"="-lboost_regex")

// boost.org/doc/libs/1_53_0/libs/regex/example/snippets/credit_card_example.cpp
#include <Rcpp.h>
#include <string>
#include <boost/regex.hpp>

bool validate_card_format(const std::string& s) {
    static const boost::regex e("(\d{4}\d{4}\d{4}\d{4})");
    return boost::regex_match(s, e);
}

// [[Rcpp::export]]
std::vector<bool> regexDemo(std::vector<std::string> s) {
    int n = s.size();
    std::vector<bool> v(n);
    for (int i=0; i<n; i++)
        v[i] = validate_card_format(s[i]);
    return v;
}
```
Plugin support in Rcpp

```r
# setup plugins environment
.plugins <- new.env()
# built-in C++11 plugin
.plugins[["cpp11"]]<- function()
if (getRversion() >= "3.1")
   list(env = list(USE_CXX1X = "yes"))
else if (.Platform$OS.type == "windows")
   list(env = list(PKG_CXXFLAGS = "-std=c++0x"))
else
   list(env = list(PKG_CXXFLAGS = "-std=c++11"))
}
# built-in OpenMP++11 plugin
.plugins[["openmp"]]<- function()
list(env = list(PKG_CXXFLAGS="-fopenmp", PKG_LIBS="-fopenmp"))
# register a plugin
registerPlugin <- function(name, plugin) {
   .plugins[[name]] <- plugin
}
```

C++11 AUTO: first-steps-with-C++11/

```c
#include <Rcpp.h>

// Enable C++11 via this plugin
[[Rcpp::plugins("cpp11")]]

// [[Rcpp::export]]
int useAuto() {
    auto val = 42;  // val will be of type int
    return val;
}
```

C++11 INITLIST: first-steps-with-C++11/

```c
#include <Rcpp.h>

// [[Rcpp::plugins("cpp11")]]

std::vector<std::string> useInitLists() {
    std::vector<std::string> vec =
    {"larry", "curly", "moe"};
    return vec;
}
```

C++11 RANGE: first-steps-with-C++11/

```c
#include <Rcpp.h>

// [[Rcpp::plugins("cpp11")]]

// [[Rcpp::export]]
int simpleProd(std::vector<int> vec) {
    int prod = 1;
    for (int &x : vec) {  // loop over all values of vec
        prod *= x;  // access each elem., comp. prod
    }
    return prod;
}
```
#include <Rcpp.h>

// [[Rcpp::plugins("cpp11")]]

// [[Rcpp::export]]
std::vector<double> transformEx(const std::vector<double>& x) {
    std::vector<double> y(x.size());
    std::transform(x.begin(), x.end(), y.begin(),
                    [](double x) { return x*x; } );
    return y;
}

Rcpp Course @ JSM 2016
149/176

// [[Rcpp::plugins("openmp")]]
#include <Rcpp.h>
// [[Rcpp::export]]
double long_computation_omp(int nb, int threads=1) {
    double sum = 0;
    #ifdef _OPENMP
        if (threads > 0) omp_set_num_threads( threads );
        Rfprintf("Number of threads=%i\n", omp_get_max_threads());
    #endif

    double thread_sum = 0;
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < nb; ++i) {
        double thread_sum = 0;
        for (int j = 0; j < nb; ++j) {
            thread_sum += R::dlnorm(i+j, 0.0, 1.0, 0);
        }
        sum += thread_sum;
    }
    return sum + nb;
}

Rcpp Course @ JSM 2016
150/176

We start we with (somewhat boring/made-up) slow double-loop:

```
#include <Rcpp.h>
// [[Rcpp::export]]
double long_computation(int nb) {
    double sum = 0;
    for (int i = 0; i < nb; ++i) {
        for (int j = 0; j < nb; ++j) {
            sum += R::dlnorm(i+j, 0.0, 1.0, 0);
        }
    }
    return sum + nb;
}
```

R> sourceCpp("code/openmpEx.cpp")
R> system.time(long_computation(1e4))
    user     system    elapsed
   22.436     0.000    22.432
R> system.time(long_computation_omp(1e4,4))
Number of threads=4
    user     system    elapsed
   25.432     0.076    7.046
R>

Even on my laptop gains can be seen:
**R Packages**

- This is an important topic in R programming
- Organising code in packages maybe *the* single most helpful step
- Core topic in R Programming / Advanced R courses
- Penn 2014 workshop had 90 minutes on this

**R Packages**

- `package.skeleton()` helpful as it creates a stanza
- `package.skeleton()` not helpful as it creates a stanza that does not pass `R CMD check` cleanly
- I wrote `pkgKitten` to provide `kitten()` which creates packages that purr
- RCpp (and RcppArmadillo, RcppEigen) all have their own versions of `package.skeleton()`
- They can use `kitten()` if `pkgKitten` is installed
- Alternative: `devtools::create()` if you don’t mind Hadleyverse dependency
- Also: RStudio File -> New Project -> New Directory -> R Package; and toggle ‘R Package’ to ‘R Package w/ Rcpp’

**Case Study: RcppAnnoy**

- Uses only one C++ header (one plus header for Windows)

```
edd@don:~/git/rcppannoy$ tree inst/include/
inst/include/
├── annoylib.h
└── mman.h
0 directories, 2 files
```
```
edd@don:~/git/rcppannoy$ tree src/
src/
├── annoy.cpp
└── Makevars
0 directories, 2 files
```
RcppAnnoy: src/Makevars

- One include indirection to the header file

```c
## We want C++11 as it gets us 'long long' as well
CXX_STD = CXX11

PKG_CPPFLAGS = -I../inst/include/
```

RcppAnnoy: src/annoy.cpp

- Implemented as Rcpp Modules (not discussed today)
  - Angular distance, or
  - Euclidian distance
- Package interesting as upstream C++ core used with Python by upstream

RcppAnnoy

- Plus a few additional files for tests and documentation.

Case Study: RcppCNPy

- Uses one C++ header and one C++ source file from CNPy

```
edd@don:~/git/rcppcnpy$ tree src/
src/
├── cnpy.cpp        # from CNPy
├── cnpy.h          # from CNPy
├── cnpyMod.cpp     # our wrapper
├── Makevars        # add -lz (from R) and C++11
└── Makevars.win    # ditto
```

edd@don:~/git/rcppcnpy$
For this package no other customization is needed
Simply add the two source files
Code integration done via Rcpp Modules (which we won’t cover today)
Here we just need one linker flag (supplied by R)

RcppCNPy: src/Makevars

One linker flag (and a compiler option for \texttt{long long})

\begin{verbatim}
## We need the compression library
PKG_LIBS = -lz
## We want C++11 as it gets us 'long long' as well
CXX_STD = CXX11
\end{verbatim}

More test files, more documentation files make this look more "busy" – but still a simple package.

Case Study: RcppAPT

RcppAPT

- A somewhat experimental package which only builds on Ubuntu or Debian
- Interface a system library we can assume to be present on those systems – but not on OS X, Windows or even other Linux systems
RcppAPT: src/Makevars

- Very simple

PKG_LIBS = -lapt-pkg

Rcpp Course @ JSM 2016

---

**Case Study: RcppFaddeeva**

- Recent package by Baptiste Auguie with some help from me
- Wrapper around some complex-valued error functions by Steven Johnson
- Upstream ships a single header and a single C++ file → just place in `src/`
- Usage pretty easy: loop over elements of argument vector and call respective function to build return vector

```
// @title Faddeeva family of error functions of the complex variable
// @description the Faddeeva function
// @param z complex vector
// @param relerr double, requested error
// @return complex vector
// @describeIn wrap compute w(z) = exp(-z^2) erfc(-iz)
// @family wrapper
// @examples
// Faddeeva_w(1:10 + 1i)
// @export
// [[Rcpp::export]]
std::vector< std::complex<double> >
    Faddeeva_w(const std::vector< std::complex<double> >& z,
                double relerr=0) {
    int N = z.size();
    std::vector< std::complex<double> > result(N);
    for(int i=0; i<N; i++) result[i] = Faddeeva::w(z[i], relerr);
    return result;
}
```
Case Study: RcppGSLExample

- This package is included in the RcppGSL package and part of the test environment.
- It implements the same column norm example we looked at earlier.

Change ‘R-only’ package to ‘R and Rcpp’ package

- No full example here
- Fairly easy to do manually:
  - Add `LinkingTo: Rcpp` to DESCRIPTION
  - Also add `Imports: Rcpp` to DESCRIPTION
  - Add `importFrom(Rcpp, "evalCpp")` to NAMESPACE
  - Add `useDynLib(yourPackageName)` to NAMESPACE
- Add some C++ code in `src/`
- Remember to run `compileAttributes()` each time you add (or change!) a C++ interface

The End
The package comes with nine pdf vignettes, and numerous help pages.

The introductory vignettes are now published (for Rcpp and RcppEigen in *J Stat Software*, for RcppArmadillo in *Comp Stat & Data Anlys*).

The rcpp-devel list is the recommended resource, generally very helpful, and fairly low volume.

StackOverflow has over 1100 posts on Rcpp too.

And a number of blog posts introduce/discuss features.