Rcpp Tutorial
Part I: Introduction

Dr. Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org

useR! 2012
Vanderbilt University
June 12, 2012

So what are we doing today?
The high-level motivation

The three main questions for the course are:

- Why? There are several reasons discussed next ...
- How? We will cover that in detail later today ...
- What? This will also be covered ...

Before the Why/How/What

Maybe some mutual introductions?

How about a really quick round of intros with

- Your name and background (academic, industry, ...)
- R experience (beginner, intermediate, advanced, ...)
- Created / modified any R packages ?
- C and/or C++ experience ?
- Main interest in **Rcpp**: speed, extension, ...,
- Following `rcpp-devel` and/or `r-devel` ?

but any such disclosure is of course strictly voluntary.

Examples

A tar file name `RcppWorkshopExamples.tar.gz` (as well as a corresponding zip file) containing all examples is at

- `http://dirk.eddelbuettel.com/code/rcpp/`
- `https://www.dropbox.com/sh/jh3fdxfd918i93n/4OxSkkKLWT`

from where you should be able to download it.

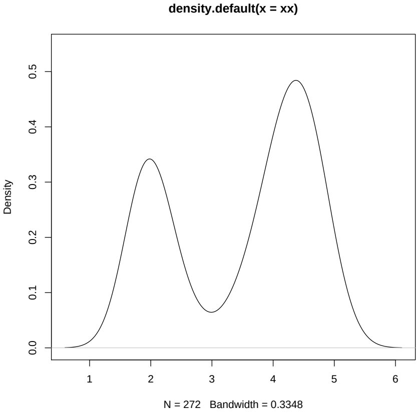
We also have copies on USB drives.

Outline

A Simple Example

Courtesy of Greg Snow via r-help during Sep 2010: examples/part1/gregEx1.R

```
xx <- faithful$eruptions
fit <- density(xx)
plot(fit)
```

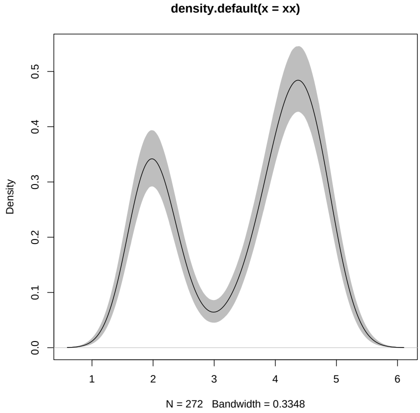


Standard R use: load some data, estimate a density, plot it.

A Simple Example

Now more complete: `examples/part1/gregEx2.R`

```
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)
```

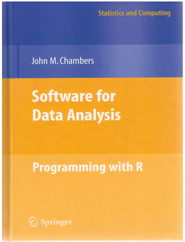


What other language can do that in seven statements?

Outline

Motivation

Why would extending R via C/C++/Rcpp be of interest?



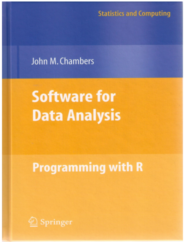
Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008

Chambers (2008) opens chapter 11 (*Interfaces I: Using C and Fortran*) with these words:

> *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.*

Motivation

Why would extending R via C/C++/Rcpp be of interest?



Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008

Chambers (2008) opens chapter 11 (*Interfaces I: Using C and Fortran*) with these words:

> *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.***

Motivation

Why would extending R via C/C++/Rcpp be of interest?

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

So the why...

The *why* boils down to:

- **speed!** Often a good enough reason for us ... and a major focus for us today.
- **new things!** We can bind to libraries and tools that would otherwise be unavailable
- **references!** Chambers quote from 2008 somehow foreshadowed the work on the new *Reference Classes* released with R 2.12 and which work very well with **Rcpp** modules. More on that this afternoon.

Outline

First speed example

examples/part1/straightCurly.R

A blog post two summers ago discussed how R's internal parser could be improved.

It repeatedly evaluated $\frac{1}{1+x}$ using

```r
## Xian's code, using <- for assignments and passing x down
f <- function(n, x=1) for (i in 1:n) x=1/(1+x)
g <- function(n, x=1) for (i in 1:n) x=(1/(1+x))
h <- function(n, x=1) for (i in 1:n) x=(1+x)^(-1)
j <- function(n, x=1) for (i in 1:n) x={1/{1+x}}
k <- function(n, x=1) for (i in 1:n) x=1/{1+x}
```

First speed example (cont.)

examples/part1/straightCurly.R

We can use this to introduce tools such as **rbenchmark**:

```
# now load some tools
library(rbenchmark)

# now run the benchmark
N <- 1e5
benchmark(f(N,1), g(N,1), h(N,1), j(N,1), k(N,1),
          columns=c("test", "replications",
                    "elapsed", "relative"),
          order="relative", replications=10)
```

First speed example (cont.)

examples/part1/straightCurly.R

```
R> N <- 1e5
R> benchmark(f(N,1),g(N,1),h(N,1),j(N,1),k(N,1),
+            columns=c("test", "replications",
+                      "elapsed", "relative"),
+            order="relative", replications=10)
     test replications elapsed relative
5 k(N, 1)           10   0.961  1.00000
1 f(N, 1)           10   0.970  1.00937
4 j(N, 1)           10   1.052  1.09469
2 g(N, 1)           10   1.144  1.19043
3 h(N, 1)           10   1.397  1.45369
R>
```

First speed example: Now with C++

examples/part1/straightCurly.R

So let us add **Rcpp** to the mix and show **inline** too:

```
## now with Rcpp and C++
library(inline)

# and define our version in C++
src <- 'int n = as<int>(ns);
        double x = as<double>(xs);
        for (int i=0; i<n; i++) x=1/(1+x);
        return wrap(x); '
l <- cxxfunction(signature(ns="integer",
                           xs="numeric"),
                 body=src, plugin="Rcpp")
```

First speed example: Now with C++

examples/part1/straightCurly.R

The key line is almost identical to what we would do in R

```
## now with Rcpp and C++
library(inline)

# and define our version in C++
src <- 'int n = as<int>(ns);
        double x = as<double>(xs);
        for (int i=0; i<n; i++) x=1/(1+x);
        return wrap(x); '
l <- cxxfunction(signature(ns="integer",
                           xs="numeric"),
                 body=src, plugin="Rcpp")
```

First speed example: Now with C++

examples/part1/straightCurly.R

Data input and output is not too hard:

```
## now with Rcpp and C++
library(inline)

# and define our version in C++
src <- 'int n = as<int>(ns);
        double x = as<double>(xs);
        for (int i=0; i<n; i++) x=1/(1+x);
        return wrap(x); '
l <- cxxfunction(signature(ns="integer",
                           xs="numeric"),
                 body=src, plugin="Rcpp")
```

First speed example: Now with C++

examples/part1/straightCurly.R

And compiling, linking and loading is a single function call:

```
## now with Rcpp and C++
library(inline)

# and define our version in C++
src <- 'int n = as<int>(ns);
        double x = as<double>(xs);
        for (int i=0; i<n; i++) x=1/(1+x);
        return wrap(x); '
l <- cxxfunction(signature(ns="integer",
                          xs="numeric"),
                 body=src, plugin="Rcpp")
```

First speed example: Now with C++

examples/part1/straightCurly.R

```
R> # now run the benchmark again
R> benchmark(f(N,1), g(N,1), h(N,1), j(N,1),
+            k(N,1), l(N,1),
+            columns=c("test", "replications",
+                      "elapsed", "relative"),
+            order="relative", replications=10)
     test replications elapsed relative
6 l(N, 1)           10   0.013   1.0000
1 f(N, 1)           10   0.944  72.6154
5 k(N, 1)           10   0.944  72.6154
4 j(N, 1)           10   1.052  80.9231
2 g(N, 1)           10   1.145  88.0769
3 h(N, 1)           10   1.425 109.6154
R>
```

First speed example: Now with C++

examples/part1/straightCurly.R

```
R> # now run the benchmark again
R> benchmark(f(N,1), g(N,1), h(N,1), j(N,1),
+            k(N,1), l(N,1),
+            columns=c("test", "replications",
+                      "elapsed", "relative"),
+            order="relative", replications=10)
     test replications elapsed relative
6 l(N, 1)           10   0.013   1.0000
1 f(N, 1)           10   0.944  72.6154
5 k(N, 1)           10   0.944  72.6154
4 j(N, 1)           10   1.052  80.9231
2 g(N, 1)           10   1.145  88.0769
3 h(N, 1)           10   1.425 109.6154
R>
```

Second speed example

examples/part1/fibonacci.R

A question on StackOverflow wondered what to do about slow recursive functions.

The standard definition of the Fibonacci sequence is $F_n = F_{n-1} + F_{n-2}$ with initial values $F_0 = 0$ and $F_1 = 1$.

This leads this intuitive (but slow) R implementation:

```
## basic R function
fibR <- function(n) {
    if (n == 0) return(0)
    if (n == 1) return(1)
    return (fibR(n - 1) + fibR(n - 2))
}
```

Second speed example: Now with C++

examples/part1/fibonacci.R

We can write an easy (and very fast) C++ version:

```
## we need a pure C/C++ function here
incltxt <- '
   int fibonacci(const int x) {
      if (x == 0) return(0);
      if (x == 1) return(1);
      return (fibonacci(x - 1)) + fibonacci(x - 2);
   }'

## Rcpp version of Fibonacci
fibRcpp <- cxxfunction(signature(xs="int"),
                       plugin="Rcpp",
                       incl=incltxt, body='
   int x = Rcpp::as<int>(xs);
   return Rcpp::wrap( fibonacci(x) );
')
```

Second speed example: Now with C++

examples/part1/fibonacci.R

So just how much faster it the C++ version?

```
R> N <- 35        ## same parameter as original post
R> res <- benchmark(fibR(N), fibRcpp(N),
+                 columns=c("test", "replications", "elapsed",
+                           "relative", "user.self", "sys.self"),
+                 order="relative", replications=1)
R> print(res)    ## show result
        test replications elapsed relative user.self sys.self
2 fibRcpp(N)            1   0.093     1.00      0.09        0
1    fibR(N)            1  61.553   661.86     61.35        0
```

So a six-hundred fold increase for no real effort or setup cost.

More on speed

Other examples:

- The **RcppArmadillo**, **RcppEigen** and **RcppGSL** packages each contain a `fastLM()` function
- This is a faster reimplementation of `lm()`, suitable for repeated use in Monte Carlo
- **Armadillo** (and **Eigen**) make this a breeze: you can do linear algebra “as you would write it with pen on paper” (but there are somewhat more technical reasons why you shouldn’t ...)
- More on that later too.

Another angle on speed

Run-time performance is just one example.

Time to code is another metric.

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

The **RcppDE** package aims to provide a concrete example of making an existing C implemention *shorter*, *easier* and possibly at the same time also *faster*. (NB: But the initial speedup may have been due to a code review – yet easier and shorter still apply.)

Outline

1. Introduction
2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References
3. How? The Tools
   - Preliminaries
   - Compiling and Linking
   - R CMD SHLIB
   - Rcpp
   - inline

Doing new things more easily

Consider the over sixty CRAN packages now using **Rcpp**. Among those, we find:

RQuantlib	QuantLib	C++
RcppArmadillo	Armadillo	C++
RcppEigen	Eigen	C++
RBrownie	Brownie (i.e. phylogenetic)	C++
RcppGSL	GNU GSL	C
RProtoBuf	(Google) Protocol Buffers	C
RSNNS	SNNS (i.e. neural nets)	C
maxent	max. entropy library (U Tokyo)	C++

A key feature is making it easy to access new functionality by making it easy to write wrappers.

Outline

S3, S4, and now Reference Classes

The new Reference Classes which appeared with R 2.12.0 are particularly well suited for multi-lingual work. C++ (via **Rcpp**) was the first example cited by John Chambers in a nice presentation at Stanford in the fall of 2010.

More in the afternoon...

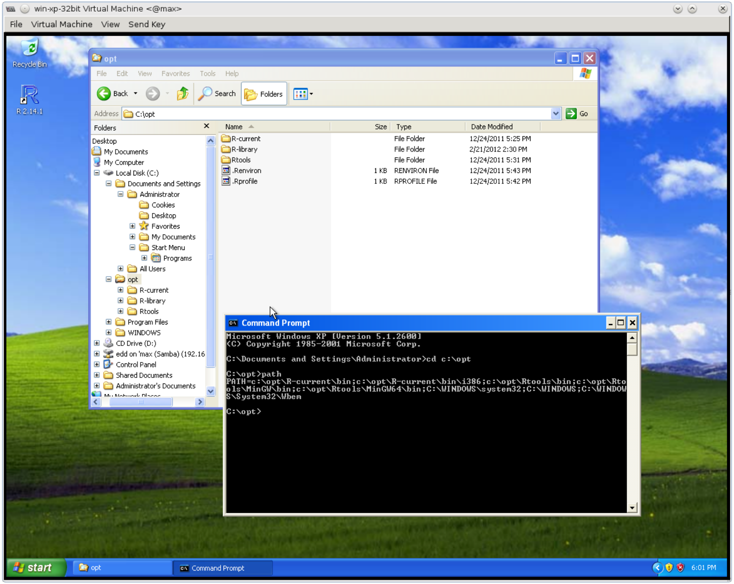
Outline

1. Introduction
2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References
3. How? The Tools
   - Preliminaries
   - Compiling and Linking
   - R CMD SHLIB
   - Rcpp
   - inline

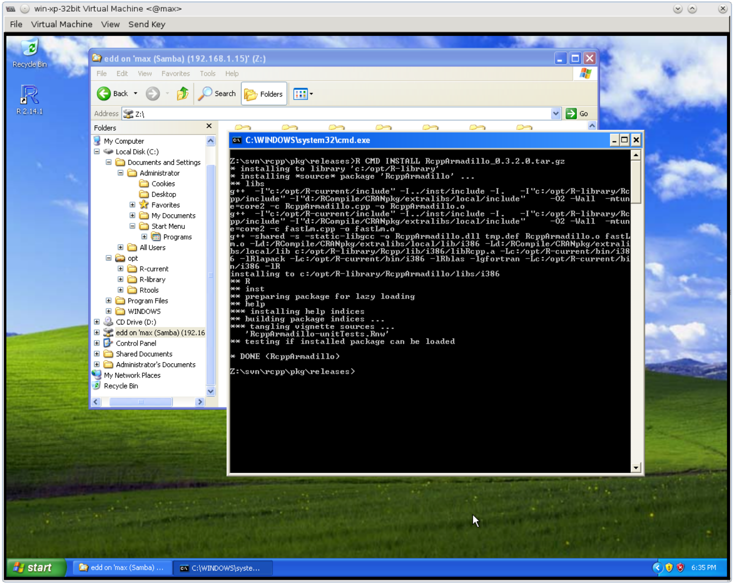
Some Preliminaries on Tools

- Use a recent version of R (>= 2.12.0 for Reference Classes; >= 2.13.0 for the R compiler package).
- Examples shown should work 'as is' on Unix-alike OSs; most will also work on Windows *provided a complete R development environment*
- *R Installation and Administration* is an excellent start to address the preceding point (if need be)
- We will compile code, so Rtools, or X Code, or standard Linux dev tools, are required.
- `using namespace Rcpp;` may be implied in some examples.

Work on Windows too – with some extra care



Work on Windows too – R CMD INSTALL as a test



Outline

1. Introduction
2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References
3. How? The Tools
   - Preliminaries
   - Compiling and Linking
   - R CMD SHLIB
   - Rcpp
   - inline

A Tradition to follow: `Hello, world!`

`examples/part1/ex1.cpp`

Let us start with some basic tool use.

Consider this simple C++ example:

```
#include <cstdio>

int main(void) {
    printf("Hello, World!\n");
}
```

A Tradition to follow: `Hello, world!`

Building and running: `examples/part1/ex1.cpp`

We can now build the program by invoking `g++`.

```
$ g++ -o ex1 ex1.cpp
$ ./ex1
Hello, World!
$
```

This use requires only one option to `g++` to select the name of the resulting *output* file.

Accessing external libraries and headers

An example using the R Math library: `examples/part1/ex2.cpp`

This example uses a function from the standalone R library :

```
#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 *declare the function via the header file* (as well as defining a variable before loading, see 'Writing R Extensions') and then need to provide a suitable *library to link to*.

Accessing external libraries and headers

An example using the R Math library: `examples/part1/ex2.cpp`

We use `-I/some/dir` to point to a header directory, and `-L/other/dir -lfoo` to link with an external library located in a particular directory.

```
$ g++ -I/usr/include -c ex2.cpp
$ g++ -o ex2 ex2.o -L/usr/lib -lRmath
$ ./ex2
N(0,1) 95th percentile 1.64485363
$
```

This can be tedious as header and library locations may vary across machines or installations. *Automated detection* is key.

Outline

Building an R module

examples/part1/modEx1.cpp

Building a dynamically callable module to be used by R is similar to the direct compilation.

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

extern "C" SEXP helloWorld(void) {
  Rprintf("Hello, World!\n");
  return R_NilValue;
}
```

Building an R module

`examples/part1/modEx1.cpp`

We use R to compile and build this:

```
$ R CMD SHLIB modEx1.cpp
g++ -I/usr/share/R/include -fpic -O3 \
      -g -c modEx1.cpp -o modEx1.o
g++ -shared -o modEx1.so \
      modEx1.o -L/usr/lib64/R/lib -lR
$
```

R can select the `-I` and `-L` flags appropriately as it knows its header and library locations.

Running the R module

examples/part1/modEx1.cpp

We load the shared library and call the function via `.Call`:

```
R> dyn.load("modEx1.so")
R> .Call("helloWorld")
Hello, World!
NULL
R>
```

Other operating systems may need a different file extension.

`R CMD SHLIB` options

`R CMD SHLIB` can take linker options.

Using the variables `PKG_CXXFLAGS` and `PKG_LIBS`, we can also select headers and libraries — which we'll look at with **Rcpp** below.

But this gets tedious fast (and example is in the next section).

Better options will be shown later.

Outline

1. Introduction
2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References
3. How? The Tools
   - Preliminaries
   - Compiling and Linking
   - R CMD SHLIB
   - Rcpp
   - inline

Rcpp and R CMD SHLIB

examples/part1/modEx2.cpp

Let us (re-)consider the first **Rcpp** example from above. In a standalone file it looks like this:

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

RcppExport SEXP modEx2(SEXP ns, SEXP xs) {
  int n = as<int>(ns);
  double x = as<double>(xs);

  for (int i=0; i<n; i++)
    x=1/(1+x);

  return wrap(x);
}
```

Rcpp and R CMD SHLIB

examples/part1/modEx2.cpp

We use PKG_CPPFLAGS and PKG_LIBS to tell R which headers and libraries. Here we let **Rcpp** tell us:

```
$ export PKG_CPPFLAGS=`Rscript -e 'Rcpp:::CxxFlags()'`
$ export PKG_LIBS=`Rscript -e 'Rcpp:::LdFlags()'`
$ R CMD SHLIB modEx2.cpp
g++ -I/usr/share/R/include \
    -I/usr/local/lib/R/site-library/Rcpp/include \
    -fpic  -O3 -pipe  -g -c modEx2.cpp -o modEx2.o
g++ -shared -o modEx2.so modEx2.o \
    -L/usr/local/lib/R/site-library/Rcpp/lib -lRcpp \
    -Wl,-rpath,/usr/local/lib/R/site-library/Rcpp/lib \
    -L/usr/lib64/R/lib -lR
```

Note the result arguments—it is helpful to understand what each part is about. Here we add the **Rcpp** library as well as information for the dynamic linker about where to find the library at run-time.

Outline

1. Introduction
2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References
3. How? The Tools
   - Preliminaries
   - Compiling and Linking
   - R CMD SHLIB
   - Rcpp
   - inline

inline

inline makes compiling, linking and loading a lot easier. As seen above, all it takes is a single call:

```
src <- 'int n = as<int>(ns);
        double x = as<double>(xs);
        for (int i=0; i<n; i++) x=1/(1+x);
        return wrap(x); '
l <- cxxfunction(signature(ns="integer",
                           xs="numeric"),
                 body=src, plugin="Rcpp")
```

No more manual `-I` and `-L` — **inline** takes over.

It also allows us to pass extra `-I` and `-L` arguments for other libraries. An (old) example using GNU GSL (which predates the **RcppGSL** package) follows:

inline – with external libraries too

examples/part1/gslRng.R

```
## a really simple C++ program calling functions from the GSL
src <- 'int seed = Rcpp::as<int>(par) ;
        gsl_rng_env_setup();
        gsl_rng *r = gsl_rng_alloc (gsl_rng_default);
        gsl_rng_set (r, (unsigned long) seed);
        double v = gsl_rng_get (r);
        gsl_rng_free(r);
        return Rcpp::wrap(v); '

## turn into a function that R can call
fun <- cfunction(signature(par="numeric"), body=src,
                 includes="#include <gsl/gsl_rng.h>",
                 Rcpp=TRUE,
                 cppargs="-I/usr/include",
                 libargs="-lgsl -lgslcblas")
```

(**RcppGSL** offers a plugin to `cxxfunction()` which alleviates four of the arguments to `cfunction` here.)

inline also good for heavily templated code

Whit's rcpp-devel post last fall: `examples/part1/whit.R`

```
library(inline)
library(Rcpp)

inc <- '
#include <iostream>
#include <armadillo>
#include <cppbugs/cppbugs.hpp>

using namespace arma;
using namespace cppbugs;

class TestModel: public MCModel {
public:
  const mat &y, &x; // given

  Normal<vec> b;
  Uniform<double> tau_y;
  Deterministic<mat> y_hat;
  Normal<mat> likelihood;
  Deterministic<double> rsq;

  TestModel(const mat& y_,const mat& X_):
    y(y_), X(X_), b(randn<vec>(X_.n_cols)),
    tau_y(1), y_hat(X*b.value),
    likelihood(y_,true), rsq(0)
  {
    add(b); add(tau_y); add(y_hat);
    add(likelihood); add(rsq);
  }
  // [....and more ...]'
```

The `inc=inc` argument to `cxxfunction` can includes headers before the `body=src` part.

And the templated `CppBUGS` package by Whit now easily outperforms PyMC / Bugs.

And is still easily accessible from R.

Outline

4 The R API

- Overview
- First Example: Operations on Vectors
- Second Example: Operations on Characters
- Third Example: Calling an R function
- Fourth Example: Creating a list

R support for C/C++

- 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: use `.cpp` extension
- R provides several calling conventions:
  - `.C()` provided the first interface, is fairly limited and no longer recommended
  - `.Call()` provides access to R objects at the C level
  - `.External()` and `.Fortran` exist but can be ignored

so we will use `.Call()` exclusively.

R API via `.Call()`

At the C level, *everything* is a `SEXP`, and all functions correspond to this interface:

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

which can be called from R via

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

and more examples will follow.

Outline

4 The R API

- Overview
- First Example: Operations on Vectors
- Second Example: Operations on Characters
- Third Example: Calling an R function
- Fourth Example: Creating a list

A simple function on vectors

examples/part1/R_API_ex1.cpp

Can you guess what this does?

```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b){
  int i, n;
  double *xa, *xb, *xab; SEXP ab;
  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  n = LENGTH(a);
  PROTECT(ab = NEW_NUMERIC(n));
  xa=NUMERIC_POINTER(a); xb=NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  double x = 0.0, y = 0.0 ;
  for (i=0; i<n; i++) xab[i] = 0.0;
  for (i=0; i<n; i++) {
    x = xa[i]; y = xb[i];
    res[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

A simple function on vectors

examples/part1/R_API_ex1.cpp

The core computation is but a part:

```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b){
  int i, n;
  double *xa, *xb, *xab; SEXP ab;
  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  n = LENGTH(a);
  PROTECT(ab = NEW_NUMERIC(n));
  xa=NUMERIC_POINTER(a); xb=NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  double x = 0.0, y = 0.0 ;
  for (i=0; i<n; i++) xab[i] = 0.0;
  for (i=0; i<n; i++) {
    x = xa[i]; y = xb[i];
    res[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

A simple function on vectors

examples/part1/R_API_ex1.cpp

Memory management is both explicit, tedious and error-prone:

```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b){
  int i, n;
  double *xa, *xb, *xab; SEXP ab;
  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  n = LENGTH(a);
  PROTECT(ab = NEW_NUMERIC(n));
  xa=NUMERIC_POINTER(a); xb=NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  double x = 0.0, y = 0.0 ;
  for (i=0; i<n; i++) xab[i] = 0.0;
  for (i=0; i<n; i++) {
    x = xa[i]; y = xb[i];
    res[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Outline

4 The R API
- Overview
- First Example: Operations on Vectors
- Second Example: Operations on Characters
- Third Example: Calling an R function
- Fourth Example: Creating a list

A simple function on character vectors

examples/part1/R_API_ex2.cpp

In R , we simply use

```
c( "foo", "bar" )
```

whereas the C API requires

```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP foobar(){
  SEXP res = PROTECT(allocVector(STRSXP, 2));
  SET_STRING_ELT( res, 0, mkChar( "foo" ) ) ;
  SET_STRING_ELT( res, 1, mkChar( "bar" ) ) ;
  UNPROTECT(1) ;
  return res ;
}
```

Outline

4 The R API

- Overview
- First Example: Operations on Vectors
- Second Example: Operations on Characters
- Third Example: Calling an R function
- Fourth Example: Creating a list

Calling an R function

examples/part1/R_API_ex2.cpp

In R , we call

```
rnorm(3L, 10.0, 20.0)
```

but in C this becomes

```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP callback(){
  SEXP call = PROTECT( LCONS( install("rnorm"),
    CONS( ScalarInteger( 3 ),
      CONS( ScalarReal( 10.0 ),
            CONS( ScalarReal( 20.0 ), R_NilValue )
          )
        )
  ) );
  SEXP res = PROTECT(eval(call, R_GlobalEnv)) ;
  UNPROTECT(2) ;
  return res ;
}
```

Outline

4 The R API
- Overview
- First Example: Operations on Vectors
- Second Example: Operations on Characters
- Third Example: Calling an R function
- Fourth Example: Creating a list

Fourth Example: Lists

examples/part1/R_API_ex4.cpp

```
#include <R.h>
#include <Rdefines.h>

extern "C" SEXP listex(){
    SEXP res = PROTECT( allocVector( VECSXP, 2 )  ) ;
    SEXP x1  = PROTECT( allocVector( REALSXP, 2 ) ) ;
    SEXP x2  = PROTECT( allocVector( INTSXP, 2 ) ) ;
    SEXP names = PROTECT( mkString( "foobar" ) ) ;

    double* px1 = REAL(x1) ; px1[0] = 0.5 ; px1[1] = 1.5 ;
    int* px2 = INTEGER(x2); px2[0] = 2 ; px2[1] = 3 ;

    SET_VECTOR_ELT( res, 0, x1 ) ;
    SET_VECTOR_ELT( res, 1, x2 ) ;
    setAttrib( res, install("class"), names ) ;

    UNPROTECT(4) ;
    return res ;
}
```

Outline

5 C++ for R Programmers

- Overview
- Compiled
- Static Typing
- Better C
- Object-Orientation
- Generic Programming and the STL
- Template Programming

C++ for R programmers

C++ is a large and sometimes complicated language.

We cannot introduce it in just a few minutes, but will will provide a number of key differences—relative to R which should be a common point of departure.

So on the next few slides, we will highlight just a few key differences, starting with big-picture differene with between R and C/C++.

One view we like comes from Meyers: *C++ is a federation of four languages.* We will also touch upon each of these four languages.

Outline

5 C++ for R Programmers

- Overview
- Compiled
- Static Typing
- Better C
- Object-Orientation
- Generic Programming and the STL
- Template Programming

Compiled rather than interpreted

We discussed this already in the context of the toolchain.

Programs need to be *compiled* first. This may require access to header files defining interfaces to other projects.

After compiling into object code, the object is *linked* into an executable, possibly together with other libraries.

There is a difference between *static* and *dynamic* linking.

Outline

5 C++ for R Programmers

- Overview
- Compiled
- Static Typing
- Better C
- Object-Orientation
- Generic Programming and the STL
- Template Programming

Static typing

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` comes close.

All these variables types are scalars which is fundamentally different from R where everything is a vector (possibly of length one).

`class` (and `struct`) allow creation of composite types; classes add behaviour to data to form *objects*.

Outline

C++ is a better C — with similarities to R

- 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 folks had with C can be avoided via STL (which is something **Rcpp** promotes too)
- that said, it is still useful to know what a pointer is ...

Outline

5 C++ for R Programmers

- Overview
- Compiled
- Static Typing
- Better C
- Object-Orientation
- Generic Programming and the STL
- Template Programming

Object-oriented programming

This is a second key feature of C++, and it does it differently from S3 and S4 (but closer to the new Reference Classes). Let's look at an example:

```
struct Date {
   unsigned int year
   unsigned int month;
   unsigned int date;
};

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

These are just nested data structures.

Object-oriented programming

OO in the C++ sense marries data with code to operate on it:

```
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();
}
```

Here the data is hidden, access to get / set is provided via an interface.

Outline

5 C++ for R Programmers
- Overview
- Compiled
- Static Typing
- Better C
- Object-Orientation
- Generic Programming and the STL
- Template Programming

Standard Template Library: Containers

The STL promotes *generic* programming via an efficient implementation.

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`.

Standard Template Library: Iterators and Algorithms

Traversal of containers can be achieved via *iterators* which require suitable member functions `begin()` and `end()`:

```
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*:

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

Other popular 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

Outline

5 C++ for R Programmers
- Overview
- Compiled
- Static Typing
- Better C
- Object-Orientation
- Generic Programming and the STL
- Template Programming

Template Programming

Template programming provides the last 'language within C++'. One of the simplest template examples is

```
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.

Template Programming

Another template example is a class squaring its argument:

```
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 some of the STL algorithms. For example, given an object `x` that has iterators, then

```
transform(x.begin(), x.end(), square);
```

squares all its elements in-place.

Rcpp Tutorial
Part II: Rcpp Details

Dr. Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org

useR! 2012
Vanderbilt University
June 12, 2012

Outline

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

RObject

The `RObject` class is the basic class behind the **Rcpp** API.

It provides a thin wrapper around a `SEXP` object—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 *you* do not have to do memory management.

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

Overview of classes: Comparison

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

Overview of key vector / matrix classes

`IntegerVector` vectors of type `integer`

`NumericVector` vectors of type `numeric`

`RawVector` vectors of type `raw`

`LogicalVector` vectors of type `logical`

`CharacterVector` vectors of type `character`

`GenericVector` generic vectors implementing `list` types

Common core functions for Vectors and Matrices

Key operations for all vectors, styled after STL 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

Outline

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

A first example

examples/part2/intVecEx1.R

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

```
library(inline)

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

Passing data from from R

examples/part2/intVecEx1.R

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

```
library(inline)

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

Objects tell us their size

examples/part2/intVecEx1.R

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

```
library(inline)

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

Element access

examples/part2/intVecEx1.R

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

```
library(inline)

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

Returning results

examples/part2/intVecEx1.R

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

```
library(inline)

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

An STL variant

examples/part2/intVecEx2.R

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

```
library(inline)

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

Outline

A first example

examples/part2/numVecEx1.R

NumericVector is very similar to IntegerVector.

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

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

A second example

Remember to `clone`: `examples/part2/numVecEx2.R`

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

A second example: continued

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

Understanding why these two examples perform differently is important:

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

Constructor overview

For `NumericVector` and other vectors deriving from `RObject`

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

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

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

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

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

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

Matrices
examples/part2/numMatEx1.R

NumericMatrix is a specialisation of NumericVector which uses a dimension attribute:

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

Matrices: RcppArmadillo for math

examples/part2/numMatEx3.R

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

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

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

Other vector types

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

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

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

`Named` can be used to assign named elements in a vector, similar to the R construct `a <- c(foo=3.14, bar=42)` letting us set attribute names (example below); `"_"` is a shortcut alternative we will see in a few examples.

Outline

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

GenericVector class (aka List) to receive values

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

```
RcppExport SEXP newRcppParamsExample(SEXP params) {

  Rcpp::List rparam(params);    // Get parameters in params.
  std::string method = Rcpp::as<std::string>(rparam["method"]);
  double tolerance   = Rcpp::as<double>(rparam["tolerance"]);
  int    maxIter     = Rcpp::as<int>(rparam["maxIter"]);
  [...]
```

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

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

GenericVector class (aka List) to return values

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

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

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

Outline

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

DataFrame class

examples/part2/dataFrameEx1.R

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

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

Outline

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

Function: First example

examples/part2/functionEx1.R

Functions are another types of SEXP object we can represent:

```
src <- '
   Function s(x) ;
   return s( 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 `s`—which we feed the second argument as well as another R expression created on the spot as `decreasing=TRUE`.

Function: Second example

examples/part2/functionEx1.R

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

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

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

While convenient, there is overhead—so we prefer functions available with 'Rcpp sugar' (discussed later).

Outline

Environments

examples/part2/environmentEx1.R

The Environment class helps us access R environments.

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

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

The environement of the (base) package **stats** is instantiated, and we access the rnorm() function from it. This is an alternative to accessing build-in functions. (But note that there is also overhead in calling R functions this way.)

Outline

S4

examples/part2/S4ex1.R

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

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

Outline

2 Extending Rcpp via `as` and `wrap`

- Introduction
- Extending wrap
- Extending as
- Example

`as()` and `wrap()`

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

They are declared as

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

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

as and wrap usage example

examples/part2/asAndWrapEx1.R

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

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

Outline

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

Extending wrap: Intrusively

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

```
#include <RcppCommon.h>

class Foo {
    public:
        Foo();

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

#include <Rcpp.h>
```

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

Extending wrap: Non-Intrusively

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

```
#include <RcppCommon.h>

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

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

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

Extending wrap: Partial specialization

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

```
#include <RcppCommon.h>

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

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

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

    }
}

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

Outline

2 Extending Rcpp via `as` and `wrap`

- Introduction
- Extending wrap
- Extending as
- Example

Extending as: Intrusively

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

```
#include <RcppCommon.h>

class Foo{
    public:
        Foo() ;

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

#include <Rcpp.h>
```

Extending as: Non-Intrusively

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

```
#include <RcppCommon.h>

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

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

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

Extending as: Partial specialization

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

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

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

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

Outline

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

The RcppBDT package wraps Boost Date_Time

A simple use case of Rcpp modules

Here, `as` and `wrap` simply convert between a `Date` representation from R and one from Boost:

```
// define template specialisations for as and wrap
namespace Rcpp {
    template <> boost::gregorian::date as( SEXP dtsexp ) {
        Rcpp::Date dt(dtsexp);
        return boost::gregorian::date(dt.getYear(), dt.getMonth(), dt.getDay());
    }

    template <> SEXP wrap(const boost::gregorian::date &d) {
        boost::gregorian::date::ymd_type ymd = d.year_month_day();      // to y/m/d struct
        return Rcpp::wrap(Rcpp::Date( ymd.year, ymd.month, ymd.day ));
    }
}
```

The header file provides both the declaration and the implementation: a simple conversion from one representation to another.

The RcppBDT package wraps Boost Date_Time

Example usage of `as` and `wrap`

Two converters provide a simple usage example:

```
// thanks to wrap() template above
Rcpp::Date date_toDate(boost::gregorian::date *d) {
    return Rcpp::wrap(*d);
}

// thanks to as
void date_fromDate(boost::gregorian::date *d, SEXP dt) {
    *d = Rcpp::as<boost::gregorian::date>(dt);
}
```

There are more examples in the (short) package sources.

Outline

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

Creating a package with Rcpp

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

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

The next few slides will show its usage.

Outline

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

Calling `Rcpp.package.skeleton()`

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

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

`Rcpp.package.skeleton` creates a file tree

```
edd@max: /tmp
File Edit View Search Terminal Help
edd@max:/tmp$ tree mypackage
mypackage
├── DESCRIPTION
├── man
│   ├── mypackage-package.Rd
│   └── rcpp_hello_world.Rd
├── NAMESPACE
├── R
│   └── rcpp_hello_world.R
├── Read-and-delete-me
└── src
    ├── Makevars
    ├── Makevars.win
    ├── rcpp_hello_world.cpp
    └── rcpp_hello_world.h

3 directories, 10 files
edd@max:/tmp$
```

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

Note that the next version of **Rcpp** will include two more `.cpp` files.

Outline

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

The C++ header file

```
#ifndef _mypackage_RCPP_HELLO_WORLD_H
#define _mypackage_RCPP_HELLO_WORLD_H

#include <Rcpp.h>

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

#endif
```

The C++ source file

```
#include "rcpp_hello_world.h"

SEXP rcpp_hello_world(){
  using namespace Rcpp ;

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

  return z ;
}
```

Outline

3 Using Rcpp in your package

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

The R file

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

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

Outline

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

The DESCRIPTION file

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

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

The NAMESPACE file

Here we use a regular expression to export all symbols.

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

Outline

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

The standard Makevars file

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

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

The Windows `Makevars.win` file

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

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

Installation and Usage

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

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

[[2]]
[1] 0 1

edd@max:/tmp$
```