Rcpp Workshop
Part I: Introduction

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

Sponsored by ASA, CTSI and PCOR
Medical College of Wisconsin
Milwaukee, WI
May 11, 2013

Outline

1 Introduction

View from 30,000 feet: 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 show of hands concerning

- Your 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` ?

Examples

Several (older) sets of workshop examples (in tar.gz and zip format) are at

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

from where you should be able to download it.

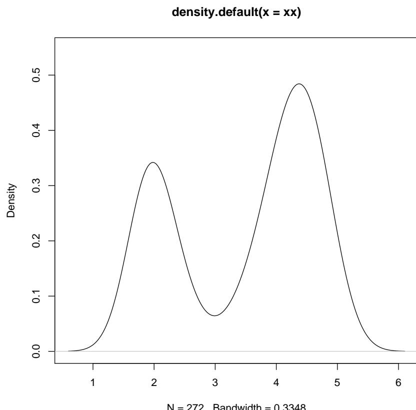
Outline

2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References

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)
```

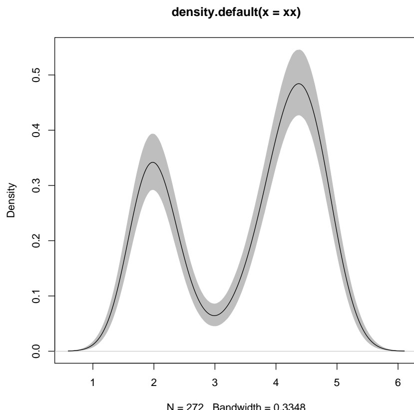


Standard R use: load some data, estimate a density, plot it.

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)
```



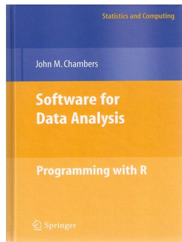
What other language can do that in seven statements?

Outline

2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References

Motivation

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



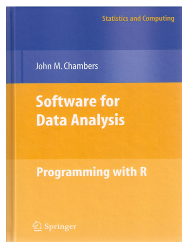
Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008

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

2. Why? The Main Motivation
   - Why R?
   - Why extend R?
   - Speed
   - New Things
   - References

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**:

```r
# 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

We can also use Rcpp attributes:

```
## Rcpp attributes version
cppFunction('
    double m(int n, double x) {
        for (int i=0; i<n; i++) x=1/(1+x);
        return x;
    }
')
```

Notice how the function is now a pure C(++) function, and all argument wrapping is done auto*magic*ally by the `cppFunction()` call.

First speed example: Now with C++

examples/part1/straightCurly.R

```
R> ## now run the benchmark again
R> N <- 1e6
R> benchmark(f(N,1), g(N,1), h(N,1), j(N,1),
+           k(N,1), l(N,1), m(N,1),
+           columns=c("test", "replications",
+                     "elapsed", "relative"),
+           order="relative", replications=10)
     test replications elapsed relative
7 m(N, 1)           10   0.105    1.000
6 l(N, 1)           10   0.133    1.267
1 f(N, 1)           10  10.866  103.486
5 k(N, 1)           10  10.871  103.533
4 j(N, 1)           10  12.368  117.790
2 g(N, 1)           10  12.758  121.505
3 h(N, 1)           10  15.694  149.467
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:

```r
## 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), fibRcppI(N), fibRcppA(N),
+                  columns=c("test", "replications",
+                            "elapsed", "relative"),
+                  order="relative",
+                  replications=1)
R> print(res)  ## show result
R>        test replications elapsed relative
2 fibRcppI(N)            1   0.101      1.0
3 fibRcppA(N)            1   0.101      1.0
1     fibR(N)            1  69.303    686.2
R>
```

A six-hundred-eighty fold increase without real effort or 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.

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.

Outline

2 Why? The Main Motivation
- Why R?
- Why extend R?
- Speed
- New Things
- References

Doing new things more easily

Consider the over one-hundred 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++
RSofia	Sofia Machine Learning Lib.	C++

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

Outline

2 Why? The Main Motivation
- Why R?
- Why extend R?
- Speed
- New Things
- References

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...

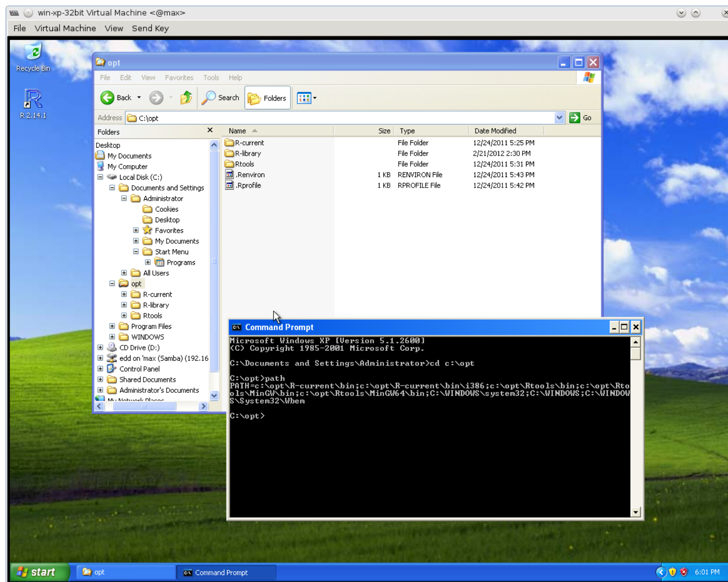
Outline

3. How? The Tools
   - Preliminaries
   - Compiling and Linking
   - R CMD SHLIB
   - Rcpp
   - inline

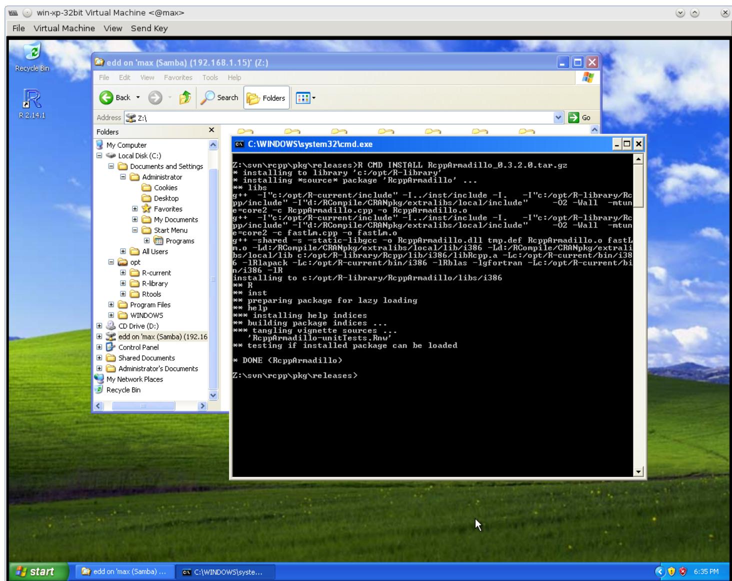
Some Preliminaries on Tools

- Use a recent version of R.
- 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.
- RStudio Server can make a lot of that go away by relying on standard Linux tools.
- `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

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

3 How? The Tools
- Preliminaries
- Compiling and Linking
- R CMD SHLIB
- Rcpp
- inline

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

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

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.)

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);
}
```

A simple function on vectors

examples/part1/R_API_ex1.cpp

But at least we can do better:

```
#include <Rcpp.h>

typedef Rcpp::NumericVector vec;

// [[Rcpp::export]]
vec vectorfoo(vec a, vec b) {
    int n = a.length();
    vec ab(n);
    for (int i=0; i<n; i++) {
        double x = a[i];
        double y = b[i];
        ab[i] = (x < y) ? x*x : -(y*y);
    }
    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

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

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.